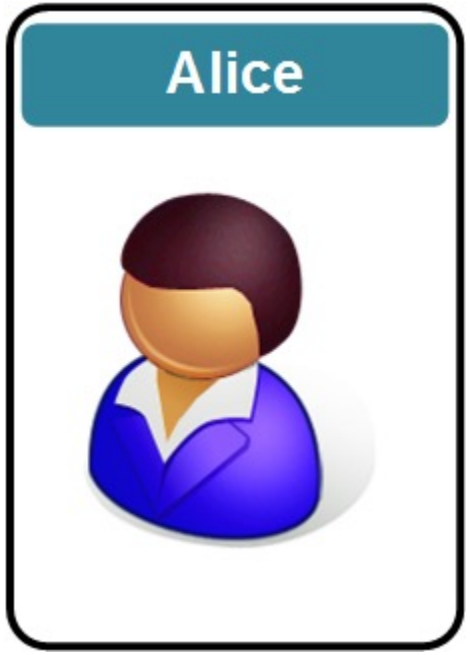
Golang and Crypto 1:

Prof Bill Buchanan OBE

https://asecuritysite.com/golang



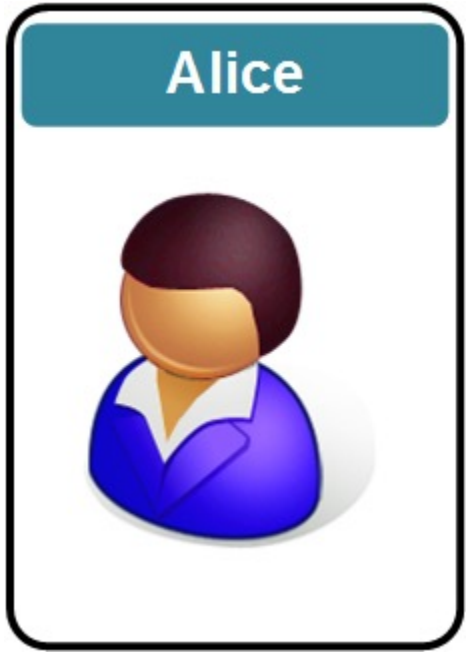
Coverage:

1. Formats: Hex, Base64 and Big Integers.
2. Symmetric Key: Block Stream and AEAD.
3. Hashing.
4. HMAC and HKDF.
5. Public Key: RSA, ElGamal and ECC.
6. Digital Signatures: ECDSA and EdDSA.

Golang and Crypto 1:

Prof Bill Buchanan OBE

https://asecuritysite.com/golang



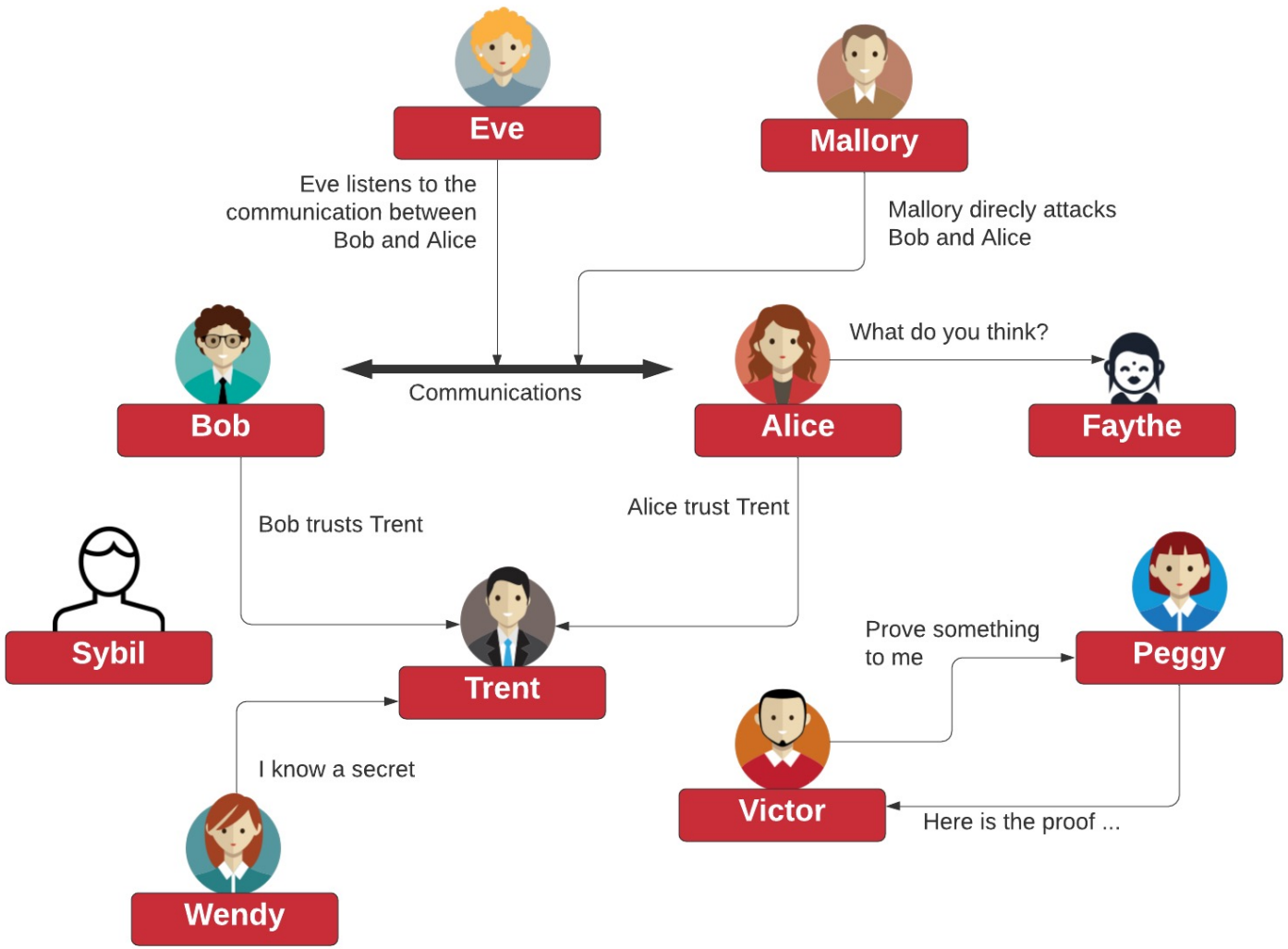
Outline:

go get github.com/test

go build

go run main.go

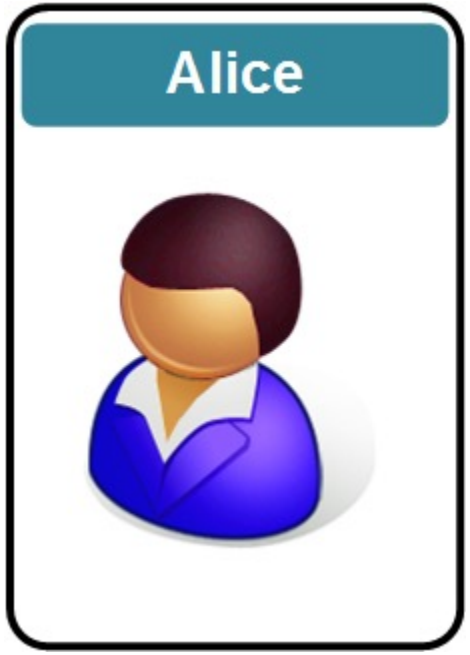
Cybersecurity actors



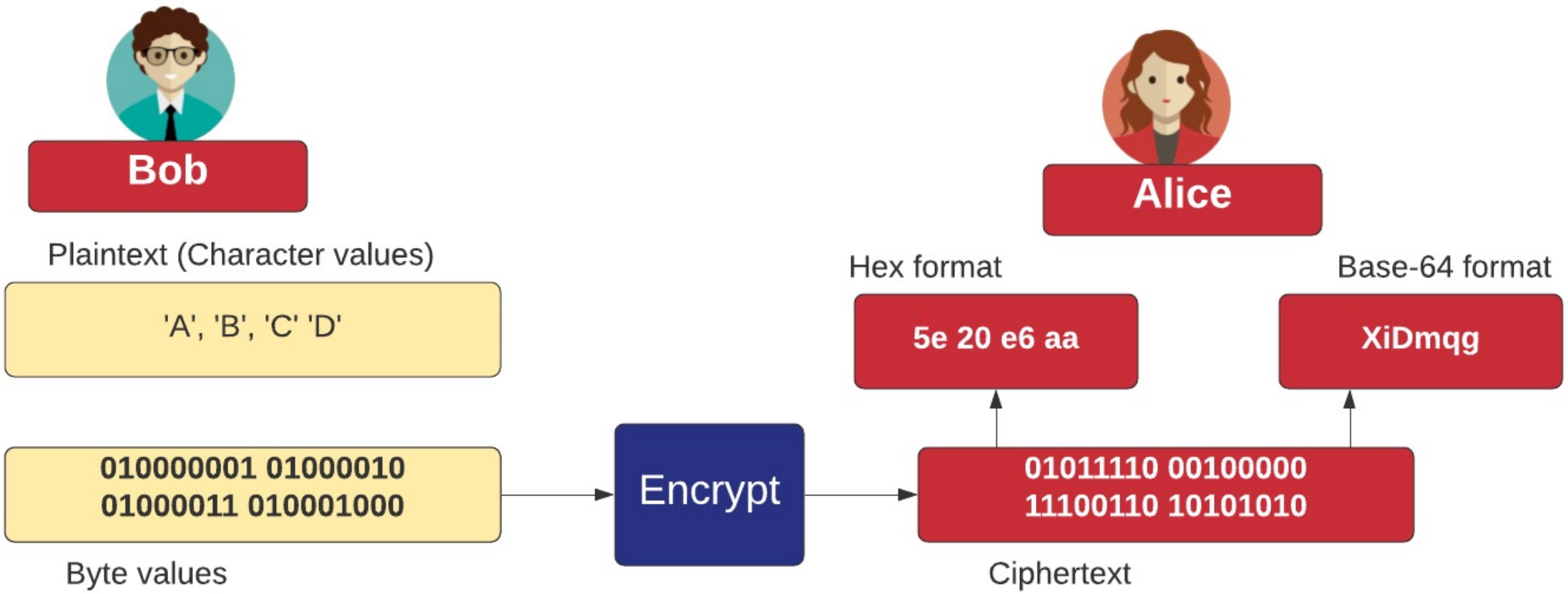
Golang and Crypto: Formats

Prof Bill Buchanan OBE

https://asecuritysite.com/golang

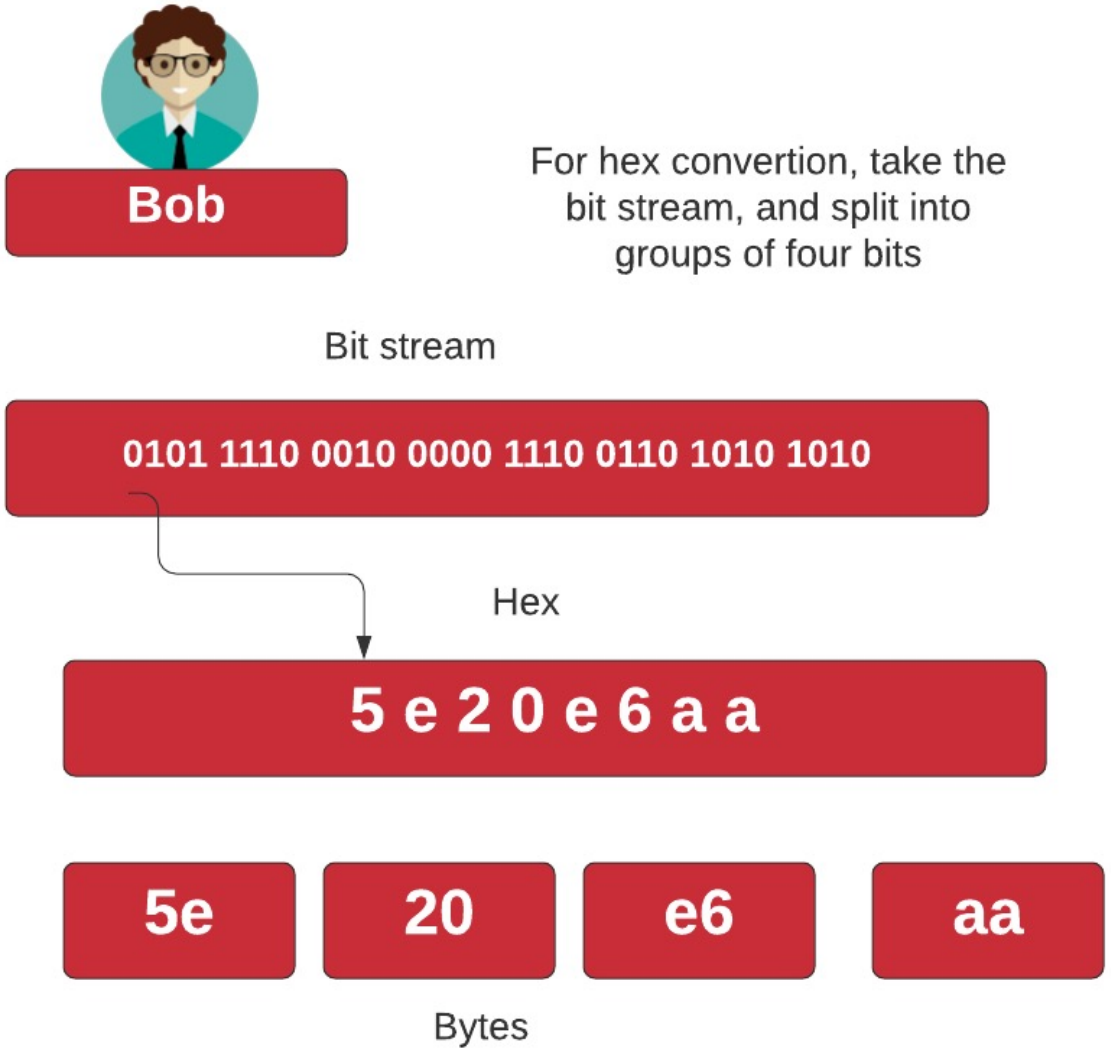


Binary Representation – Hex and Base64



https://asecuritysite.com/golang/go_base

Binary Representation – Hex



Value	Bit value
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

https://asecuritysite.com/golang/go_base

Binary Representation – Base64

Bob

For Base-64, we take the bit stream and convert into groups of six bits

Bit stream

01100110 01110010 01100101 01100100

011001 100111 001001 100101 011001 00

Base64

Z n J l Z A = =

Val	Enc	Val	Enc	Val	Enc	Val	Enc
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

https://asecuritysite.com/golang/go_base

Binary Representation – Base64

Bob

For Base-64, we take the bit stream and convert into groups of six bits

Val	Enc	Val	Enc	Val	Enc	Val	Enc
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2

Bit stream

01100110 01110010 01100101 01100100

011001 100111 001001 100101 0110

Base64

Z n J l Z A = =

```go
hash := md5.Sum([]byte(s))
hash2 := sha256.Sum256([]byte(s))

//h1.Write([]byte(s))

fmt.Printf("Input:\t\t\t%s\n", s)
fmt.Printf("Input (hex):\t\t%x\n", []byte(s))
fmt.Printf("Input (hex):\t\t%s\n", hex.EncodeToString([]byte(s)))
fmt.Printf("Input (Base64):\t\t%s\n", base64.StdEncoding.EncodeToString([]byte(s)))
fmt.Printf("MD5 (Hex):\t\t%x\n", hash)
fmt.Printf("MD5 (hex):\t\t%s\n", hex.EncodeToString(hash[:]))
fmt.Printf("MD5 (Base64):\t\t%s\n", base64.StdEncoding.EncodeToString(hash[:]))
fmt.Printf("SHA-256 (Hex):\t\t%x\n", hash2)
fmt.Printf("SHA-256 (hex):\t\t%s\n", hex.EncodeToString(hash2[:]))
fmt.Printf("SHA-256 (Base64):\t%s\n", base64.StdEncoding.EncodeToString(hash2[:]))
```

https://asecuritysite.com/golang/go_base

Binary Representation – Big Integers

Key size	No of keys
16	65,536
32	4,294,967,296
48	281,474,976,710,656
64	18,446,744,073,709,551,616
80	1,208,925,819,614,629,174,706,176
96	79,228,162,514,264,337,593,543,950,336
112	5,192,296,858,534,827,628,530,496,329,220,096
128	340,282,366,920,938,463,463,374,607,431,768,211,456
144	22,300,745,198,530,623,141,535,718,272,648,361,505,980,416
160	1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,97 6
176	95,780,971,304,118,053,647,396,689,196,894,323,976,171,195,136,4 75,136
192	6,277,101,735,386,680,763,835,789,423,207,666,416,102,355,444,46 4,034,512,896
208	411,376,139,330,301,510,538,742,295,639,337,626,245,683,966,408, 394,965,837,152,256
224	26,959,946,667,150,639,794,667,015,087,019,630,673,637,144,422,5 40,572,481,103,610,249,216
240	1,766,847,064,778,384,329,583,297,500,742,918,515,827,483,896,87 5,618,958,121,606,201,292,619,776
256	115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665, 640,564,039,457,584,007,913,129,639,936
272	7,588,550,360,256,754,183,279,148,073,529,370,729,071,901,715,04 7,420,004,889,892,225,542,594,864,082,845,696

https://asecuritysite.com/golang/bigint

Binary Representation – Big Integers

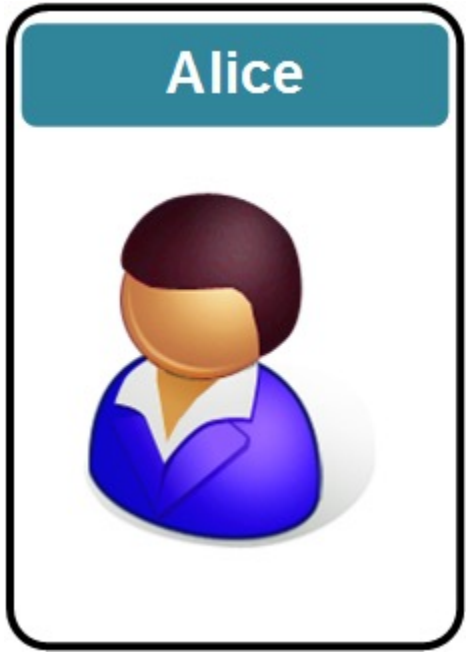
Key size	No of keys
16	65,536
32	4,294,967,296
48	281,474,976,710,656
64	18,446,744,073,709,551,616
80	1,208,925,819,614,629,174,706,176
96	79,228,162,514,264,337,593,543,950,336
112	5,192,296,858,534,827,628,530,496,329,220,096
128	340,282,366,920,938,463,463,374,607,431,768,211,456
144	22,300,745,198,530,623,141,535,718,272,648,361,505,980,416
160	1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,97 6
176	95,780,971,304,118,053,647,396,689,196,894,323,976,171,195,136,4 75,136
192	6,277,101,735,386,680,763,835,789,423,207,666,416,102,355,444,46 4,034,512,896
208	411,376,139,330,301,510,538,742,295,639,337,626,245,683,966,408, 394,965,837,152,25
224	26,959,946,667,150,639,794,667,015,087,019,630,673,637,144,422,5 40,572,481,103,610
240	1,766,847,064,778,384,329,583,297,500,742,918,515,827,483,896,87 5,618,958,121,606,
256	115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665, 640,564,039,457,58
272	7,588,550,360,256,754,183,279,148,073,529,370,729,071,901,715,04 7,420,004,889,892,

```go
x1 := toBigInt(v1)
x2 := toBigInt(v2)
prime := toBigInt(p1)
fmt.Printf("Value 1 in hex: %x\n", v1)
fmt.Printf("Value 1 in decimal: %s\n", v1)
fmt.Printf("Value 2 in hex: %x\n", v1)
fmt.Printf("Value 2 in decimal: %s\n", v1)

fmt.Printf("v1+v2= %s\n", new(big.Int).Add(x1, x2).String())
fmt.Printf("v1-v2= %s\n", new(big.Int).Sub(x1, x2).String())
fmt.Printf("v1*v2= %s\n", new(big.Int).Mul(x1, x2).String())
fmt.Printf("\n\n== Now using a prime number of 2^255-19 ===\n")


fmt.Printf("p== %s\n", prime)
p := new(big.Int).Exp(x1, x2, prime)
fmt.Printf("v1^v2 (mod p)= %s\n", prime)

p = new(big.Int).Mul(x1, x2)
p = new(big.Int).Mod(p, prime)
```

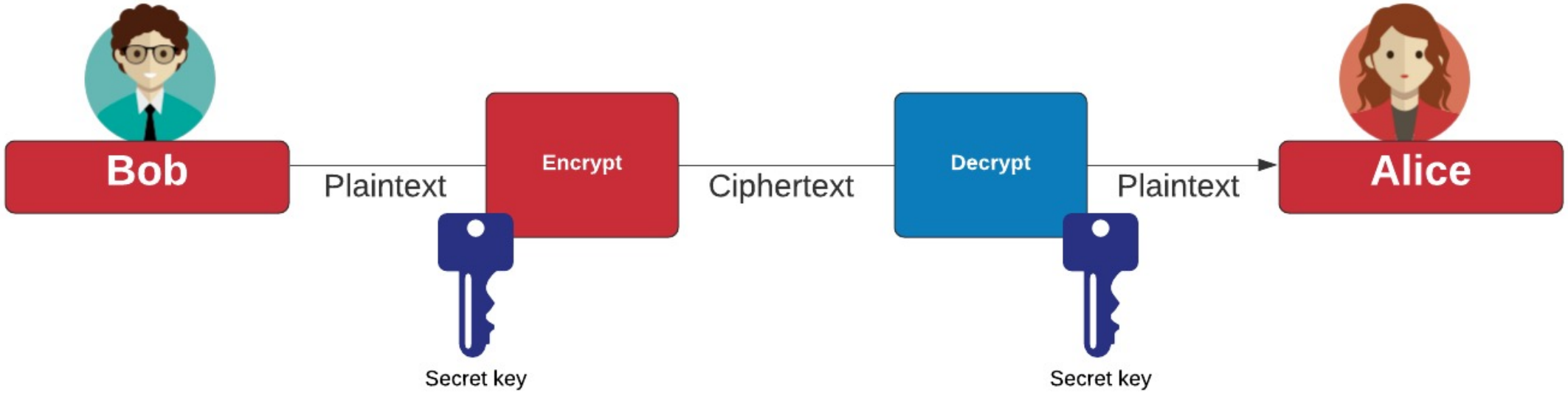
https://asecuritysite.com/golang/bigint

Golang and Crypto: Symmetric Key

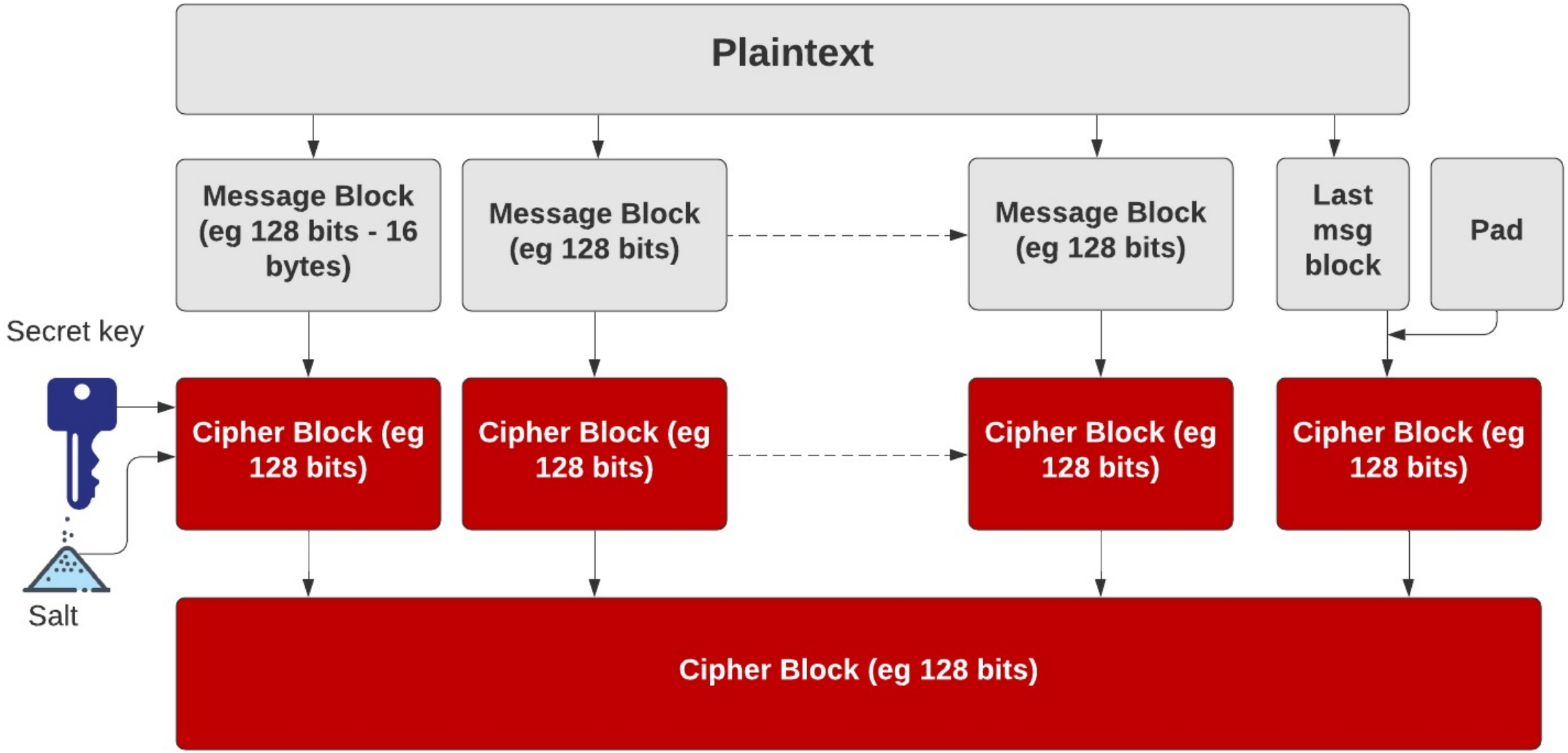
Prof Bill Buchanan OBE
https://asecuritysite.com/golang



Symmetric Key



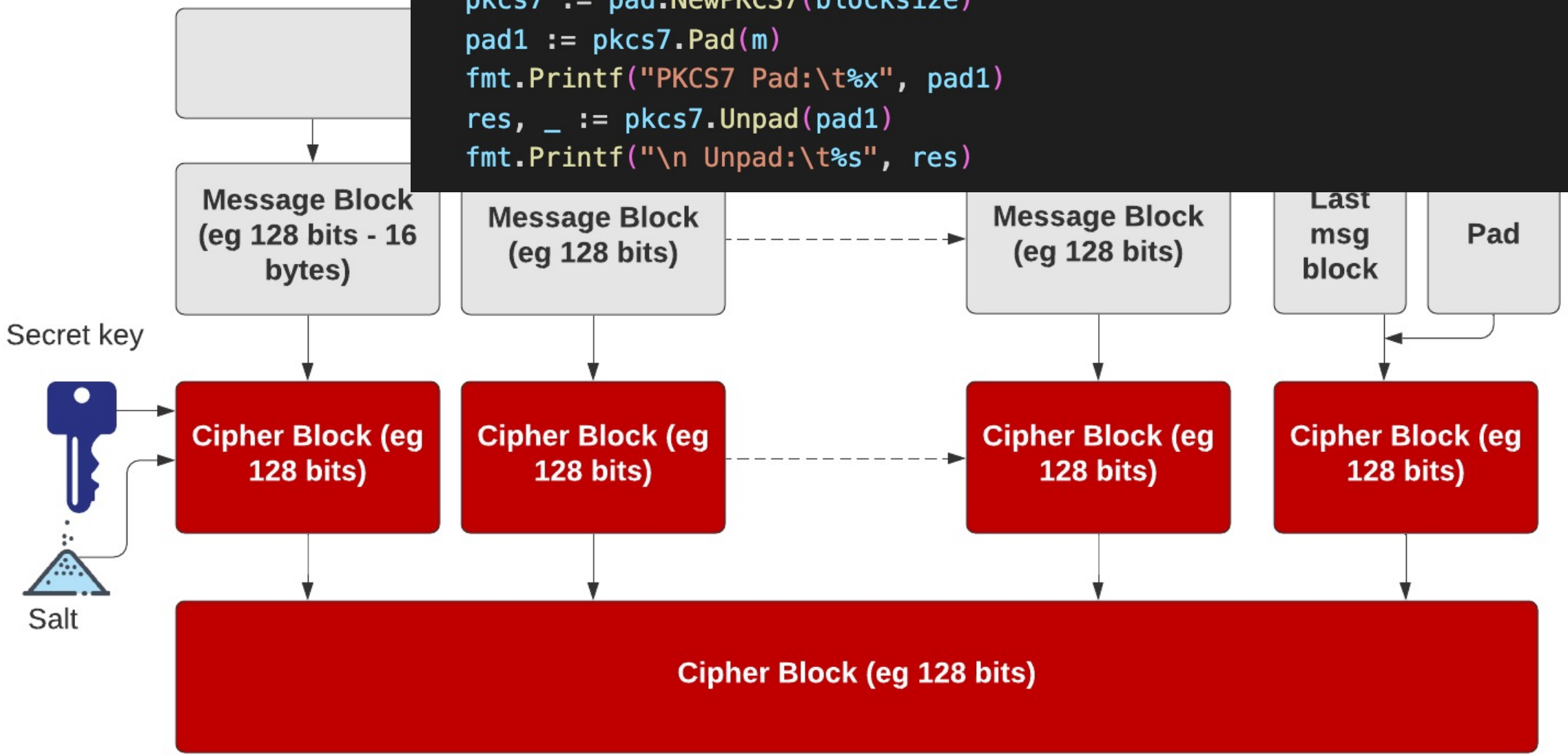
Symmetric Key – Block cipher



https://asecuritysite.com/golang/go_pad

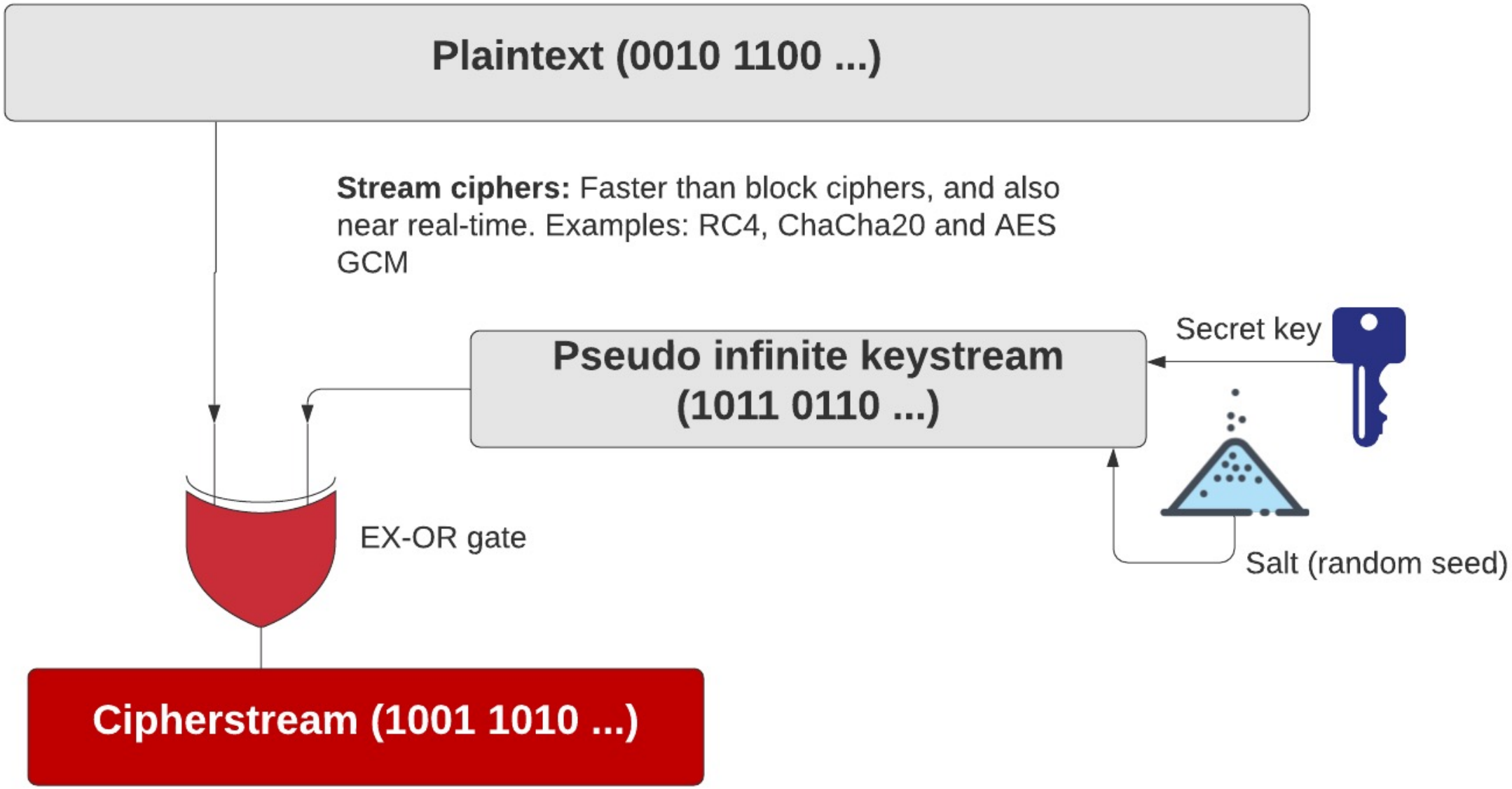
Symmetric Key –

```go
m := []byte(msg)
fmt.Printf("Block size:\t%d bytes (%d bits)\n\n", blocksize, blocksize*8)

pkcs7 := pad.NewPKCS7(blocksize)
pad1 := pkcs7.Pad(m)
fmt.Printf("PKCS7 Pad:\t%x", pad1)
res, _ := pkcs7.Unpad(pad1)
fmt.Printf("\n Unpad:\t%s", res)
```



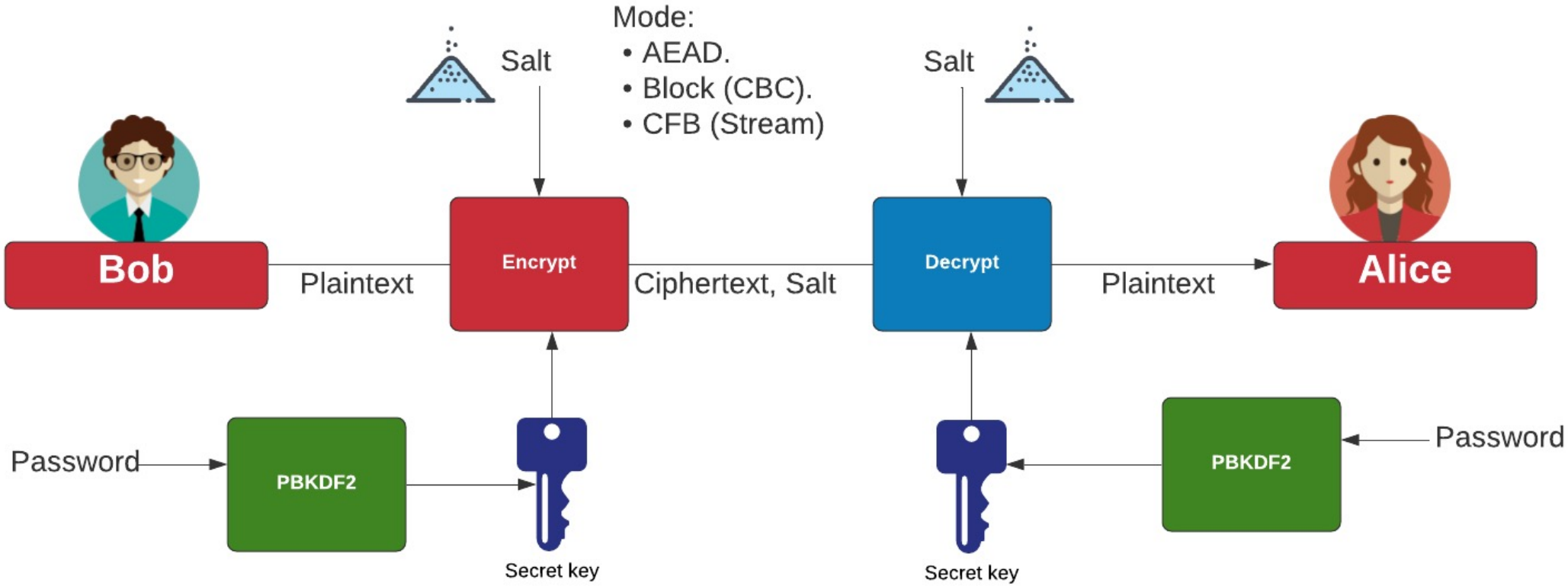
https://asecuritysite.com/golang/go_pad

Symmetric Key – Stream cipher



https://asecuritysite.com/golang/go_chacha

Symmetric Key - Modes

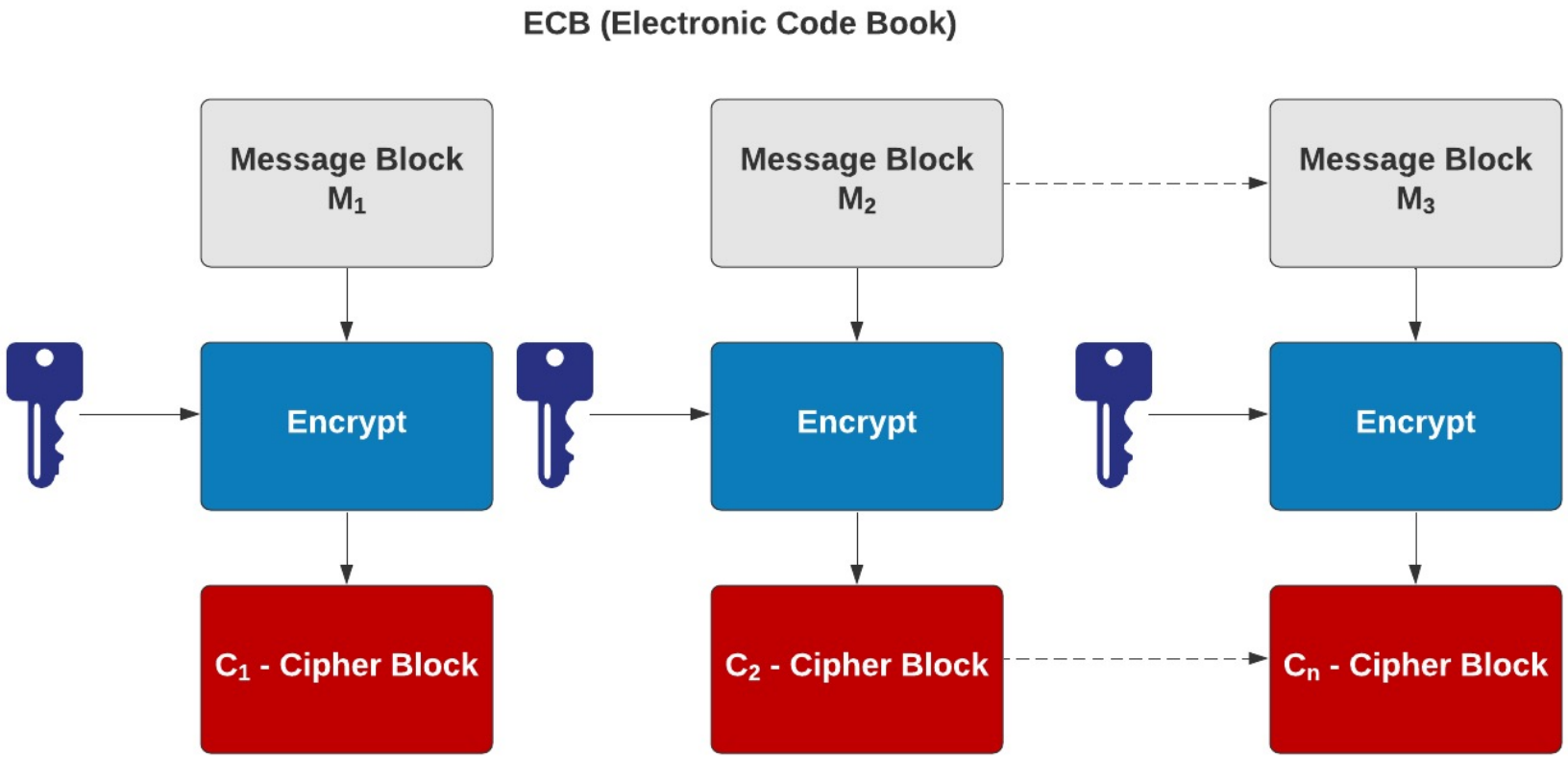


AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

https://asecuritysite.com/golang/go_symmetric

Symmetric Key – Electronic Code Code (ECB)

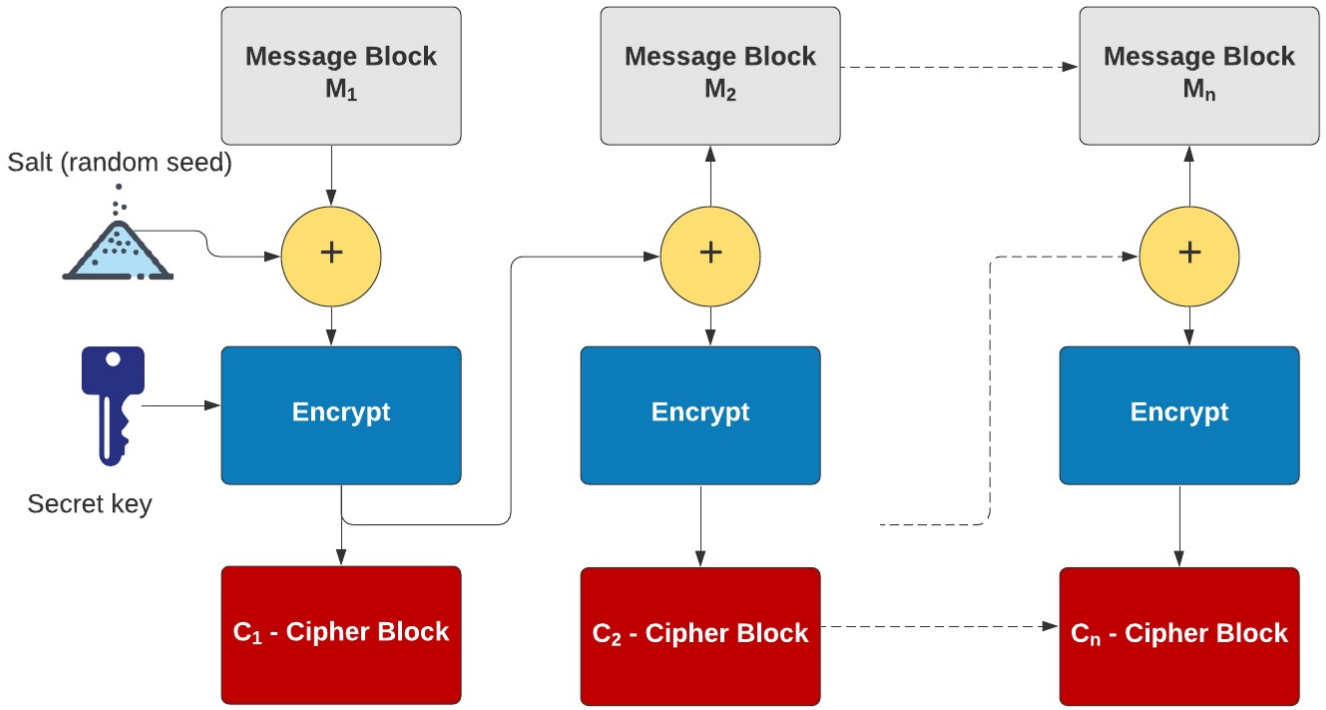


AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

https://asecuritysite.com/golang/go_symmetric

Symmetric Key – Cipher Block Chaining (CBC)

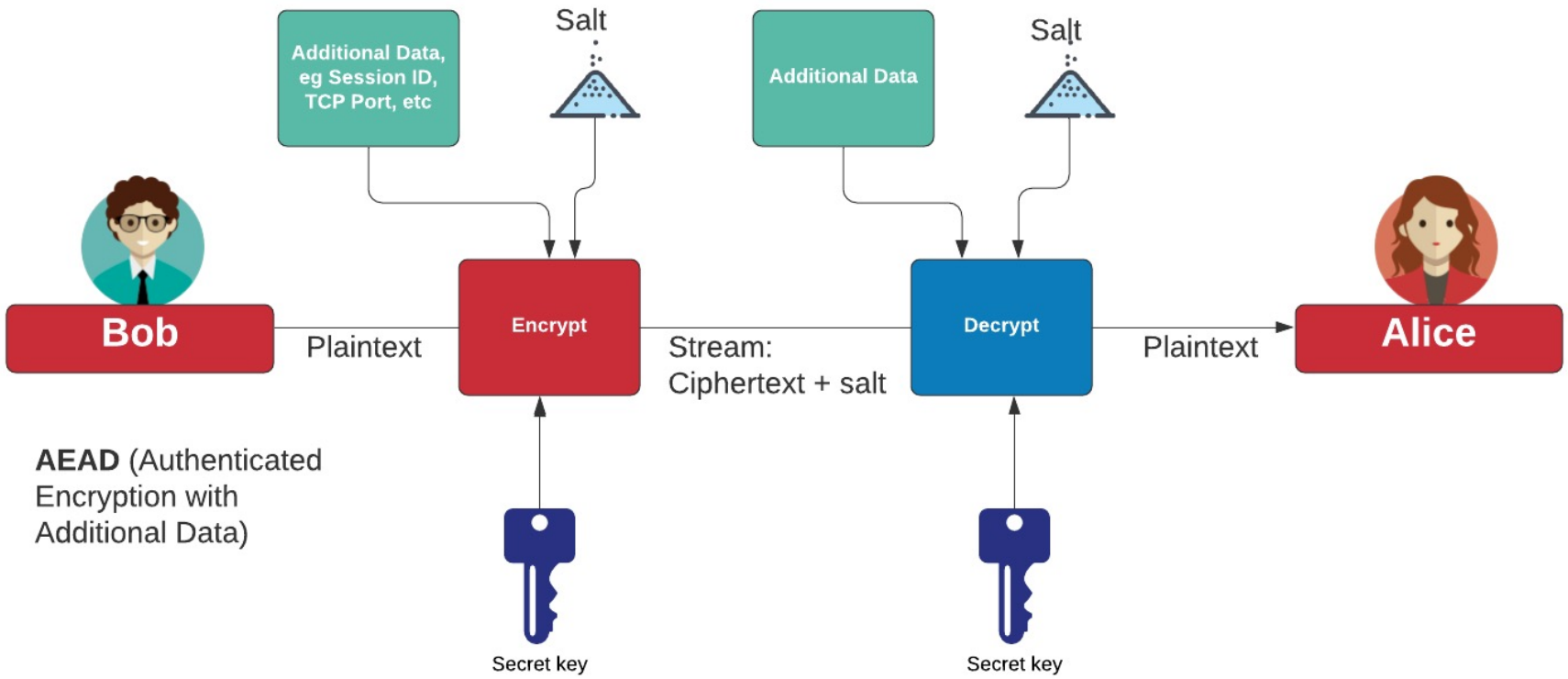


AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

https://asecuritysite.com/golang/go_symmetric

Authenticated Encryption with Additional Data (AEAD)

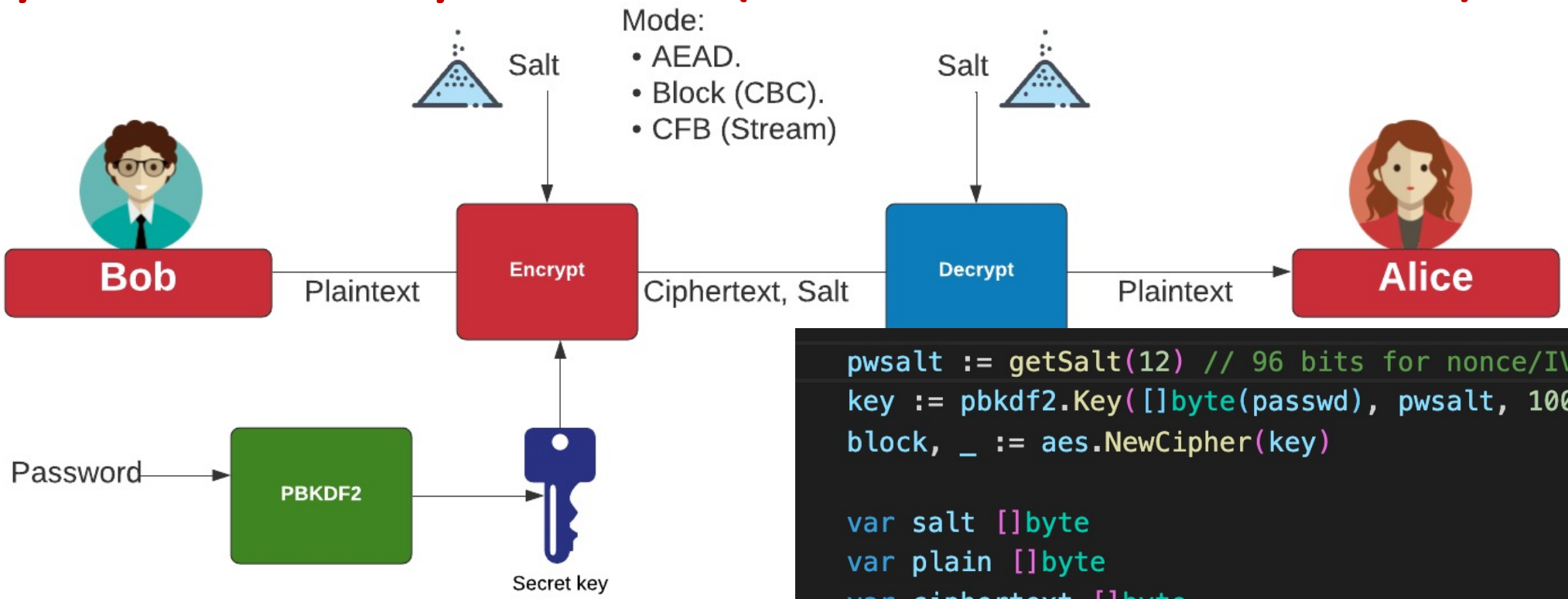


AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

https://asecuritysite.com/golang/go_symmetric

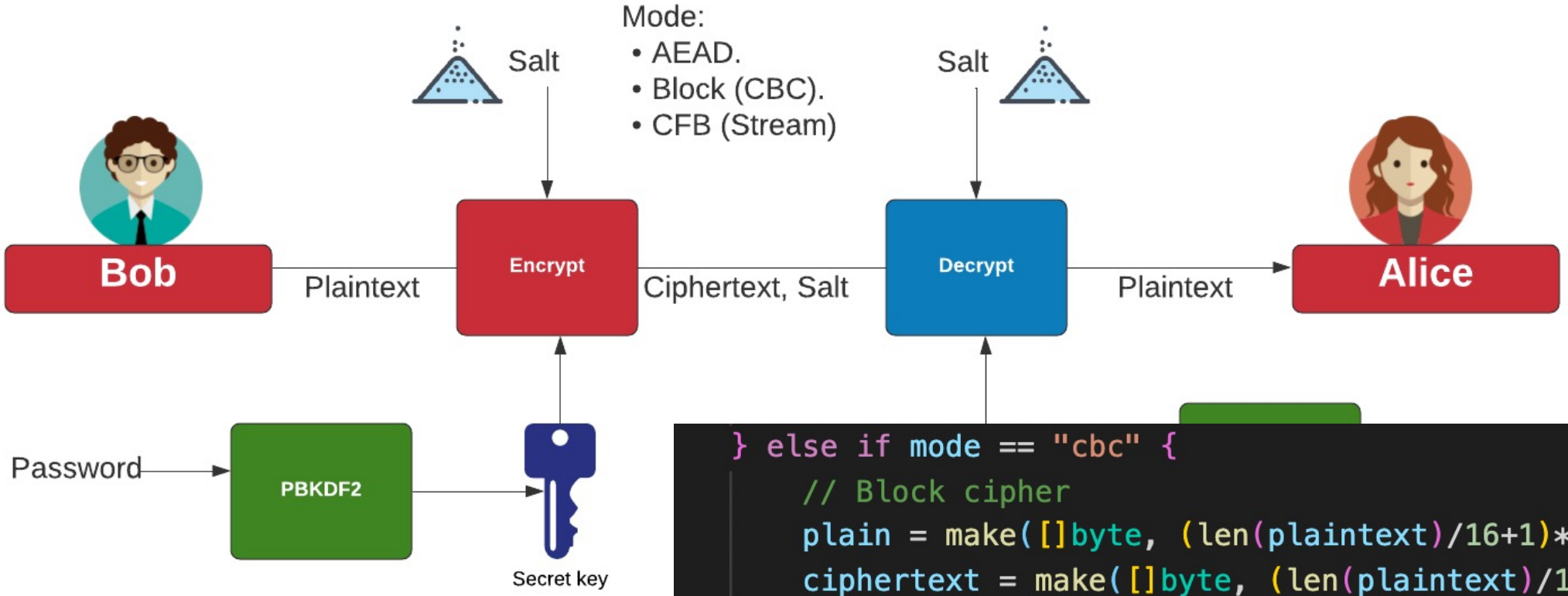
Symmetric Key – GCM (Galois Counter Mode)




```go
pwsalt := getSalt(12) // 96 bits for nonce/IV
key := pbkdf2.Key([]byte(passwd), pwsalt, 10000, size, sha256.New)
block, _ := aes.NewCipher(key)

var salt []byte
var plain []byte
var ciphertext []byte

plaintext := []byte(msg)

if mode == "gcm" {
    // AEAD
    salt = getSalt(12)
    aesgcm, _ := cipher.NewGCM(block)
    ciphertext = aesgcm.Seal(nil, salt, plaintext, nil)
    plain, _ = aesgcm.Open(nil, salt, ciphertext, nil)
```

https://asecuritysite.com/golang/go_symmetric

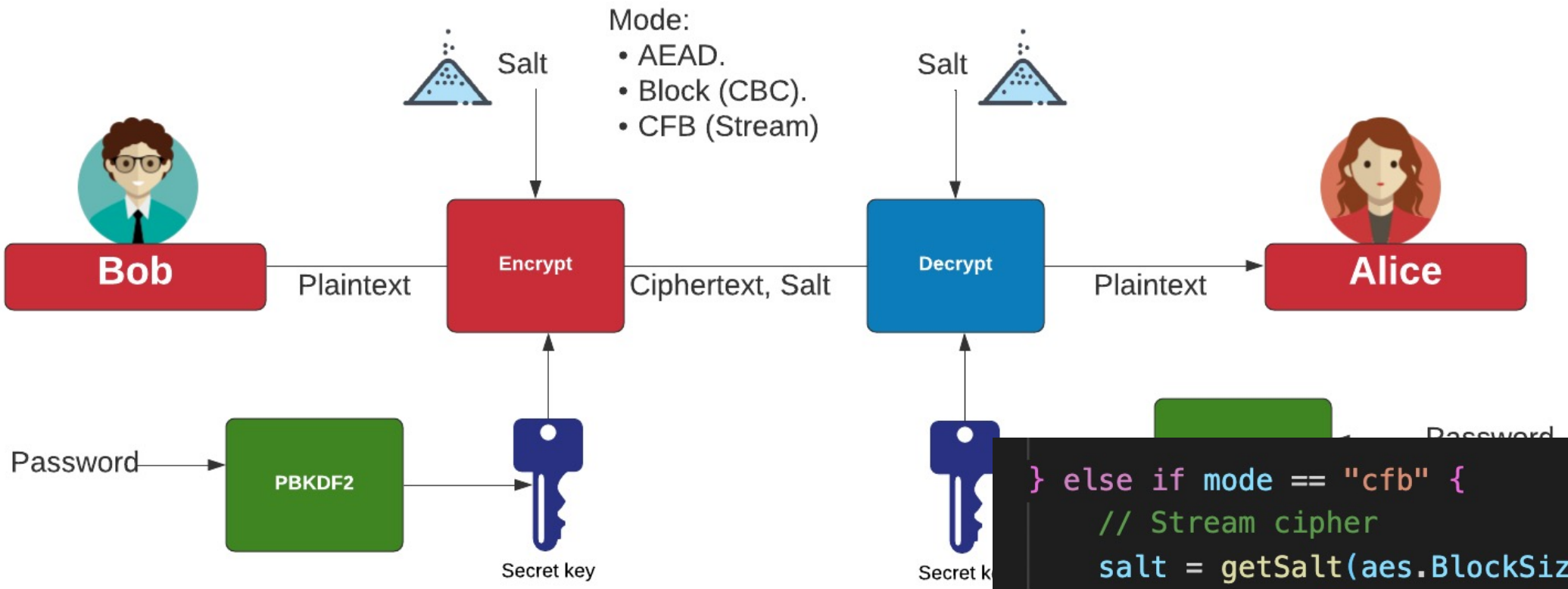
Symmetric Key – Block Cipher (CBC)




```
} else if mode == "cbc" {
    // Block cipher
    plain = make([]byte, (len(plaintext)/16+1)*aes.BlockSize)
    ciphertext = make([]byte, (len(plaintext)/16+1)*aes.BlockSize)
    salt = getSalt(16)
    pkcs7 := pad.NewPKCS7(aes.BlockSize)
    pad1 := pkcs7.Pad(plaintext)
    blk := cipher.NewCBCEncrypter(block, salt)
    blk.CryptBlocks(ciphertext, pad1)
    blk = cipher.NewCBCDecrypter(block, salt)
    blk.CryptBlocks(plain, ciphertext)
    plain, _ = pkcs7.Unpad(plain)
```

https://asecuritysite.com/golang/go_symmetric

Symmetric Key – Stream Cipher (CFB)



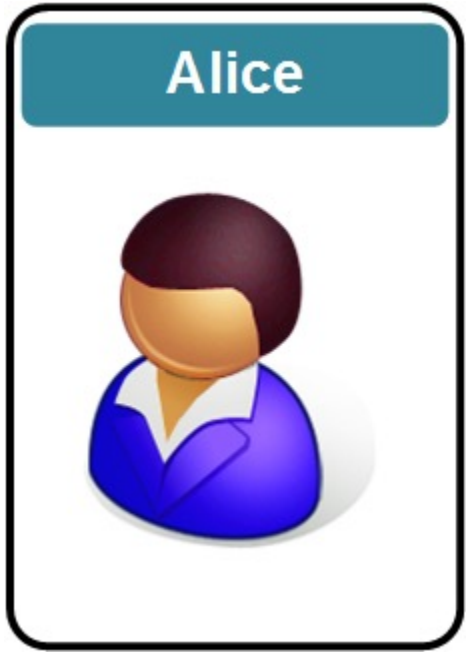

```
} else if mode == "cfb" {
    // Stream cipher
    salt = getSalt(aes.BlockSize)
    plain = make([]byte, len(plaintext))
    ciphertext = make([]byte, len(plaintext))

    stream := cipher.NewCFBEncrypter(block, salt)
    stream.XORKeyStream(ciphertext, plaintext)
    stream = cipher.NewCFBDecrypter(block, salt)
    stream.XORKeyStream(plain, ciphertext)
}
```

https://asecuritysite.com/golang/go_symmetric

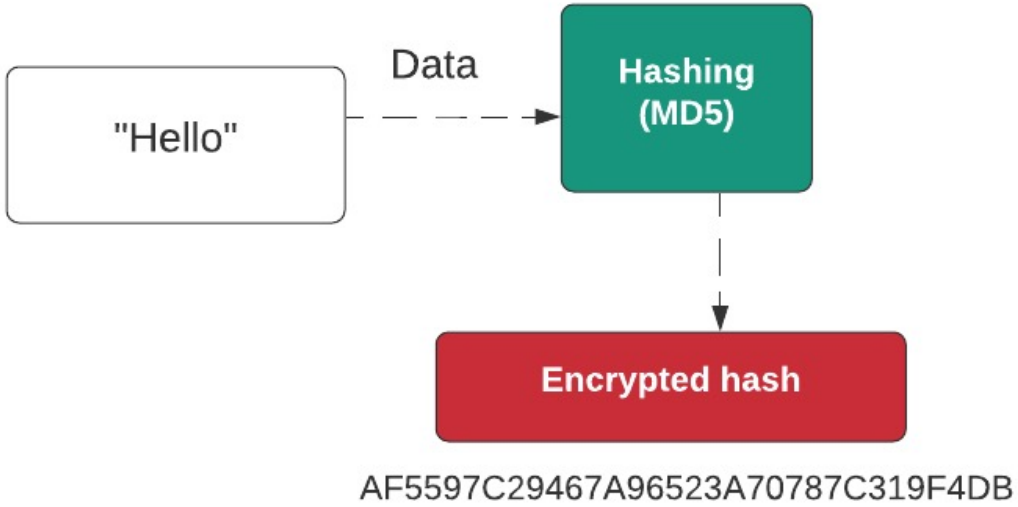
Golang and Crypto: Hashing

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com/golang



Hashing



Hashed password

Bob: AF5597C29467A96523A70787C319F4DB

Hashed password with salt (hellozj8n)

Bob: zj8n:51A7C663A3BDCD06D6CE21E2BCB2AD5A

https://asecuritysite.com/golang/gohash

Hashing – MD5 (128-bit)

Hex

hello	5D41402ABC4B2A76B9719D911017C592
Hello	8B1A9953C4611296A827ABF8C47804D7
How are you?	04E35EB3E4FCB8B395191053C359CA0E
Napier	8F83571F9324AE4E23D773753055C7B6

Base64

hello	XUFAKrxLKna5cZ2REBfFkg==
Hello	ixqZU8RhEpaoJ6v4xHgE1w==
How are you?	BONes+T8uLOVGRBTw1nKDg==
Napier	j4NXH5Mkrk4j13N1MFXHtg==

https://asecuritysite.com/golang/gohash

Hashing – SHA-1 (160-bit)

Hex

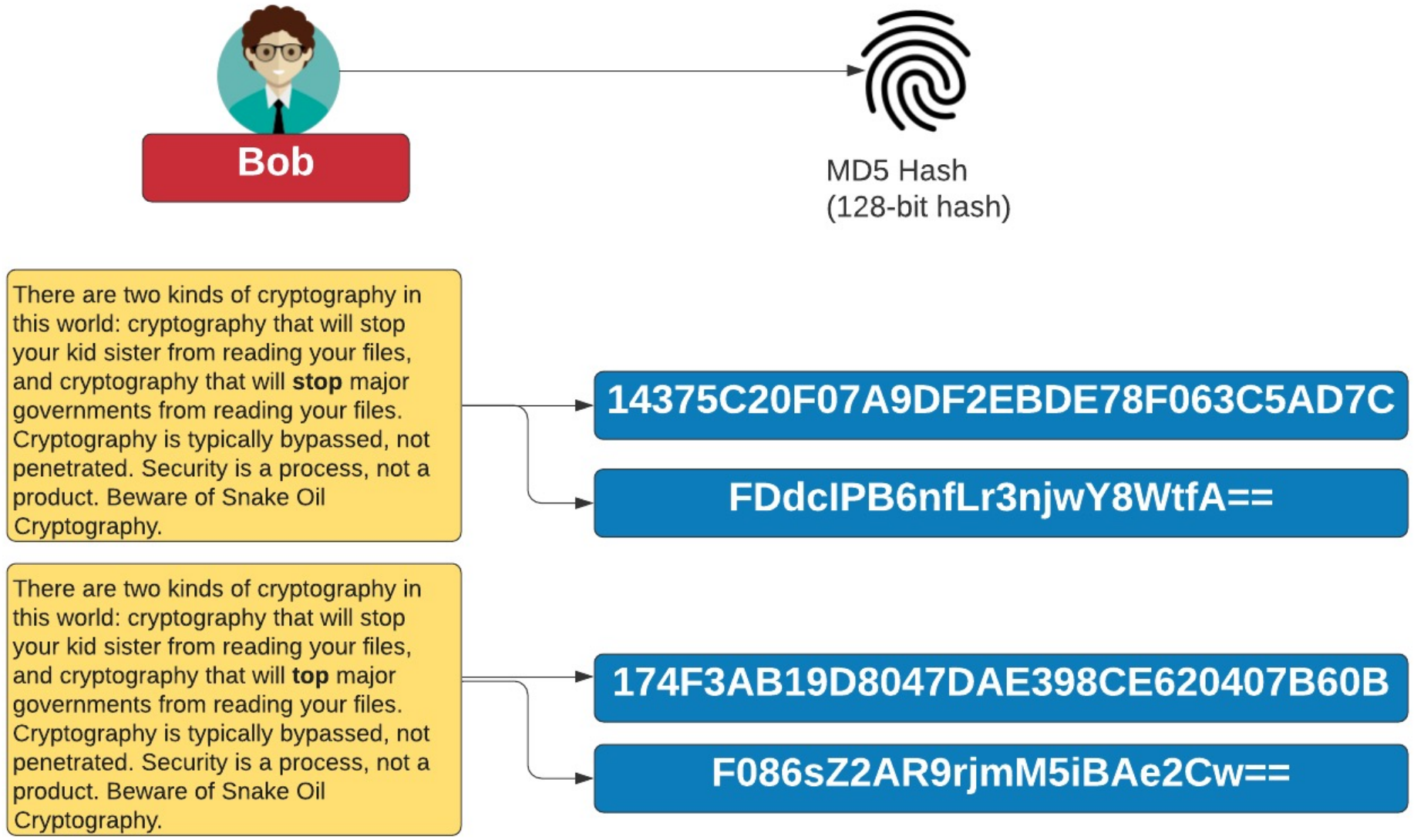
hello	AAF4C61DDCC5E8A2DABEDE0F3B482CD9AEA9434D
Hello	F7FF9E8B7BB2E09B70935A5D785E0CC5D9D0ABF0
How are you?	3031897E282167593FBB4DBE81DC48EBBE9A002D
Napier	BF81B135A5687766F4F464764EAC38CA8A4EBABA

Base64

hello	qvTGHdzF6KLavt4PO0gs2a6pQ00=
Hello	9/+ei3uy4Jtwk1pdeF4MxdnQq/A=
How are you?	MDGJfighZ1k/u02+gdxI676aAC0=
Napier	v4GxNaVod2b09GR2Tqw4yopOuro=

https://asecuritysite.com/golang/gohash

Hashing



https://asecuritysite.com/golang/gohash

Hashing

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **stop** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

14375C20F07A9DF2EE

FDdcIPB6nfLr3r

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **top** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

174F3AB19D8047DAE

F086sZ2AR9rjmM

```go
s := "abc"

h1 := md4.New()
h2 := md5.Sum([]byte(s))
h3 := sha1.Sum([]byte(s))
h4 := sha256.Sum256([]byte(s))
h5 := sha512.Sum512([]byte(s))
h6 := ripemd160.New()
h7 := sha3.Sum256([]byte(s))
h8 := sha3.Sum512([]byte(s))
h9 := blake2b.Sum256([]byte(s))
h10 := blake2b.Sum512([]byte(s))
```

https://asecuritysite.com/golang/gohash

Hashing and OpenSSL

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **stop** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

14375C20F07A9DF2EBDE78F063C5AD7C

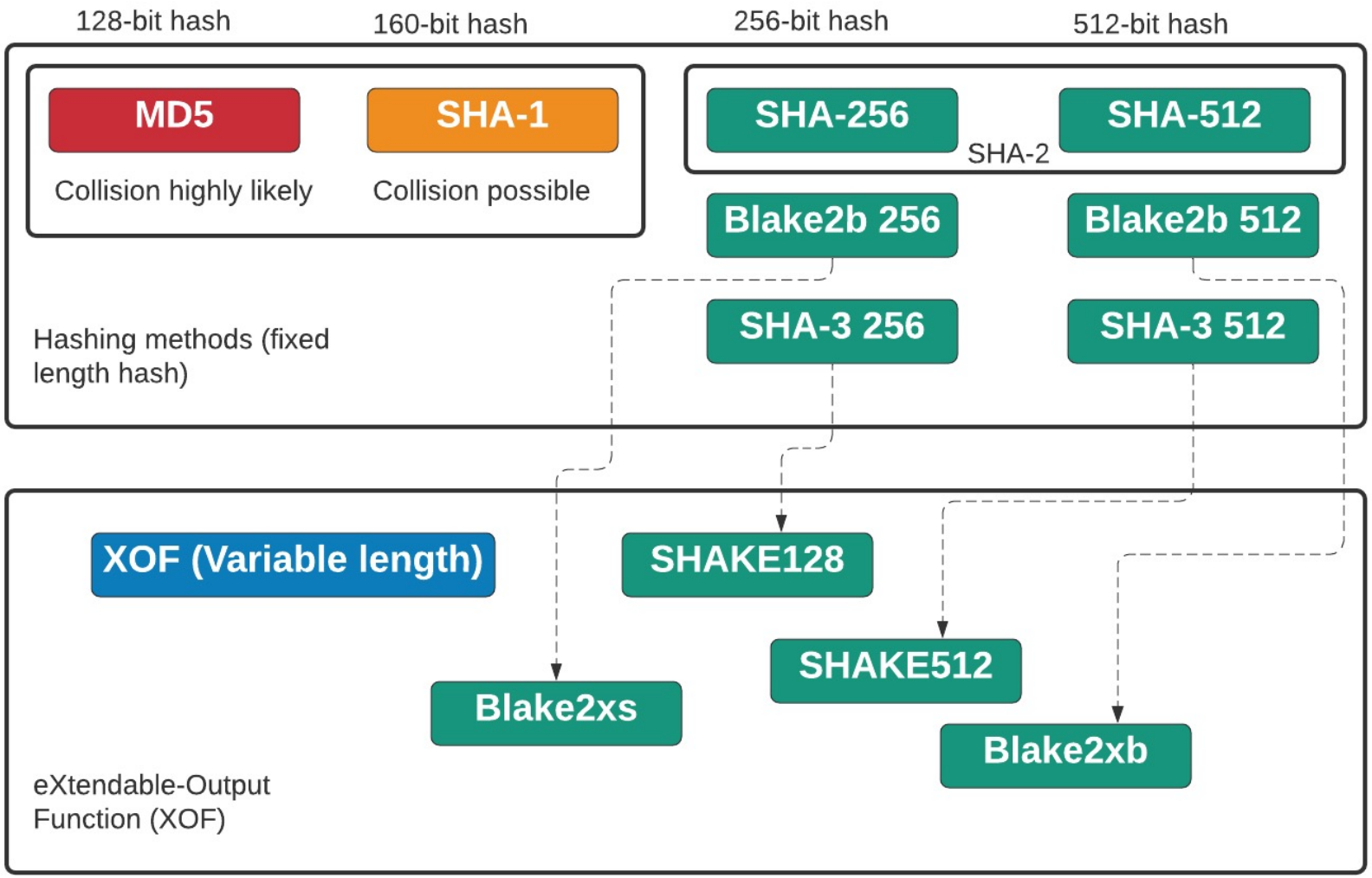
FDdcIPB6nfLr3njwY8WtfA==

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will **top** major governments from reading your files. Cryptography is typically bypassed, not penetrated. Security is a process, not a product. Beware of Snake Oil Cryptography.

```
Linux command: echo -n "Hello" | openssl dgst -md5
Windows command: echo | set /p = "Hello" | openssl dgst -md5

Message: Hello
Mode: md5
========
8b1a9953c4611296a827abf8c47804d7
```

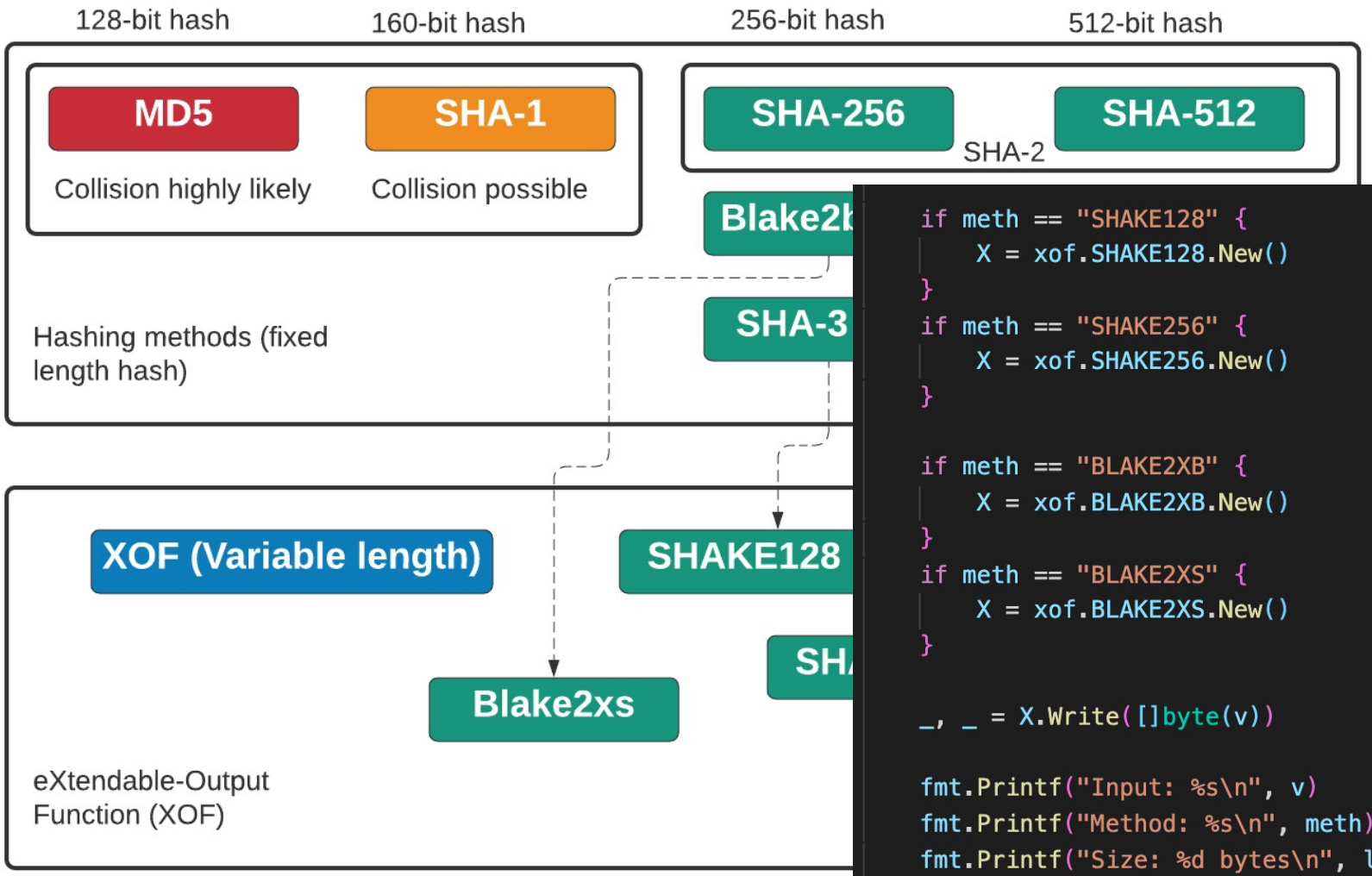
https://asecuritysite.com/golang/gohash

MD5, SHA-1, SHA-2, SHA-3 and XOF



https://asecuritysite.com/golang/circl_xof

MD5, SHA-1, SHA-2, SHA-3 and XOF



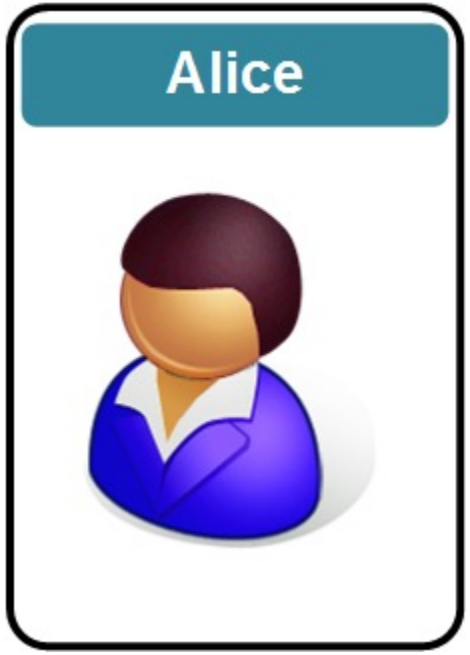

```
if meth == "SHAKE128" {
    X = xof.SHAKE128.New()
}
if meth == "SHAKE256" {
    X = xof.SHAKE256.New()
}

if meth == "BLAKE2XB" {
    X = xof.BLAKE2XB.New()
}
if meth == "BLAKE2XS" {
    X = xof.BLAKE2XS.New()
}

_, _ = X.Write([]byte(v))

fmt.Printf("Input: %s\n", v)
fmt.Printf("Method: %s\n", meth)
fmt.Printf("Size: %d bytes\n", length)
xof_out := make([]byte, length)

_, _ = X.Read(xof_out)
fmt.Printf("\nHash: %x\n", xof_out[:length])
```

https://asecuritysite.com/golang/circl_xof

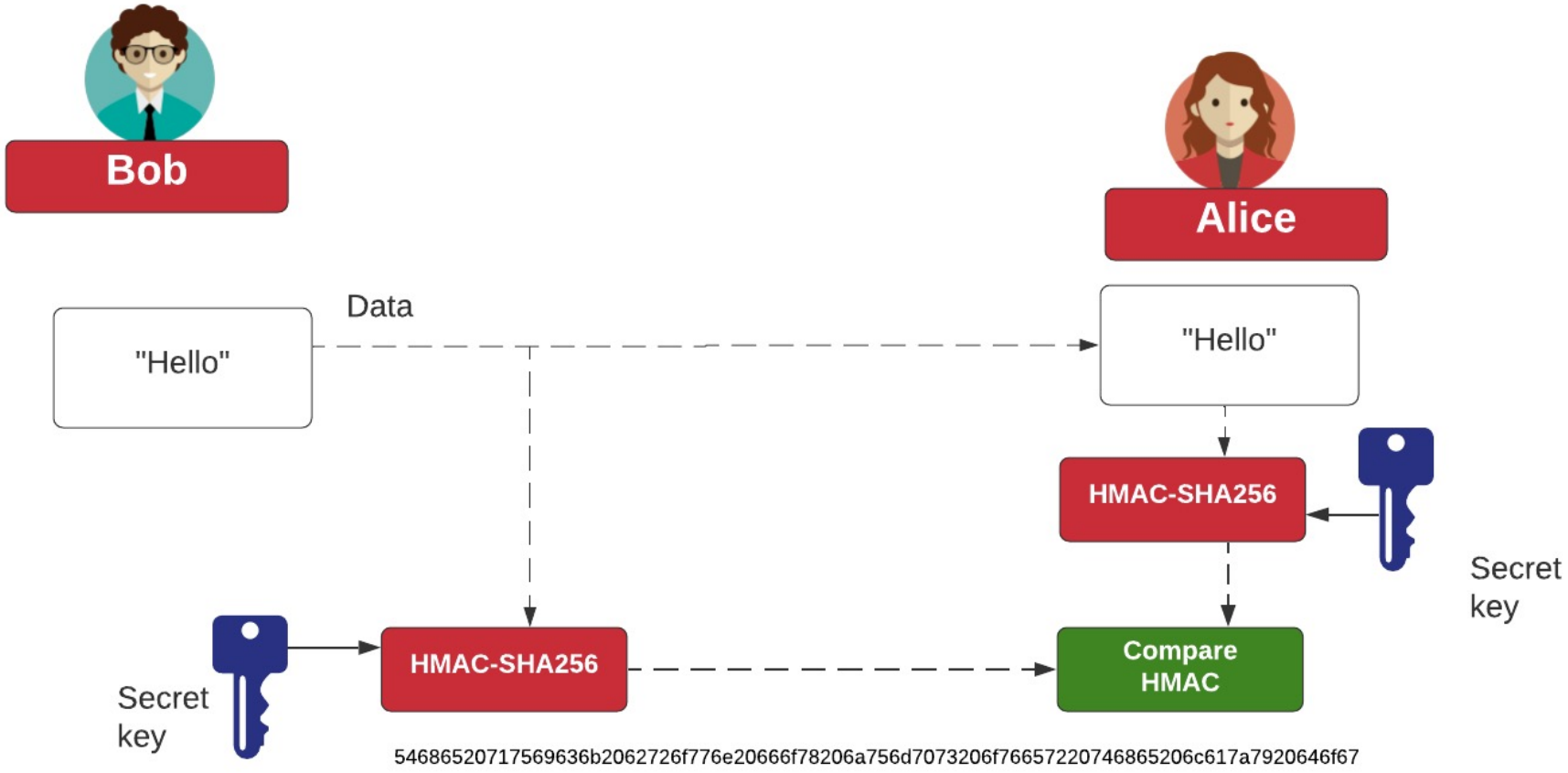
Golang and Crypto: HMAC and HKDF

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com/golang

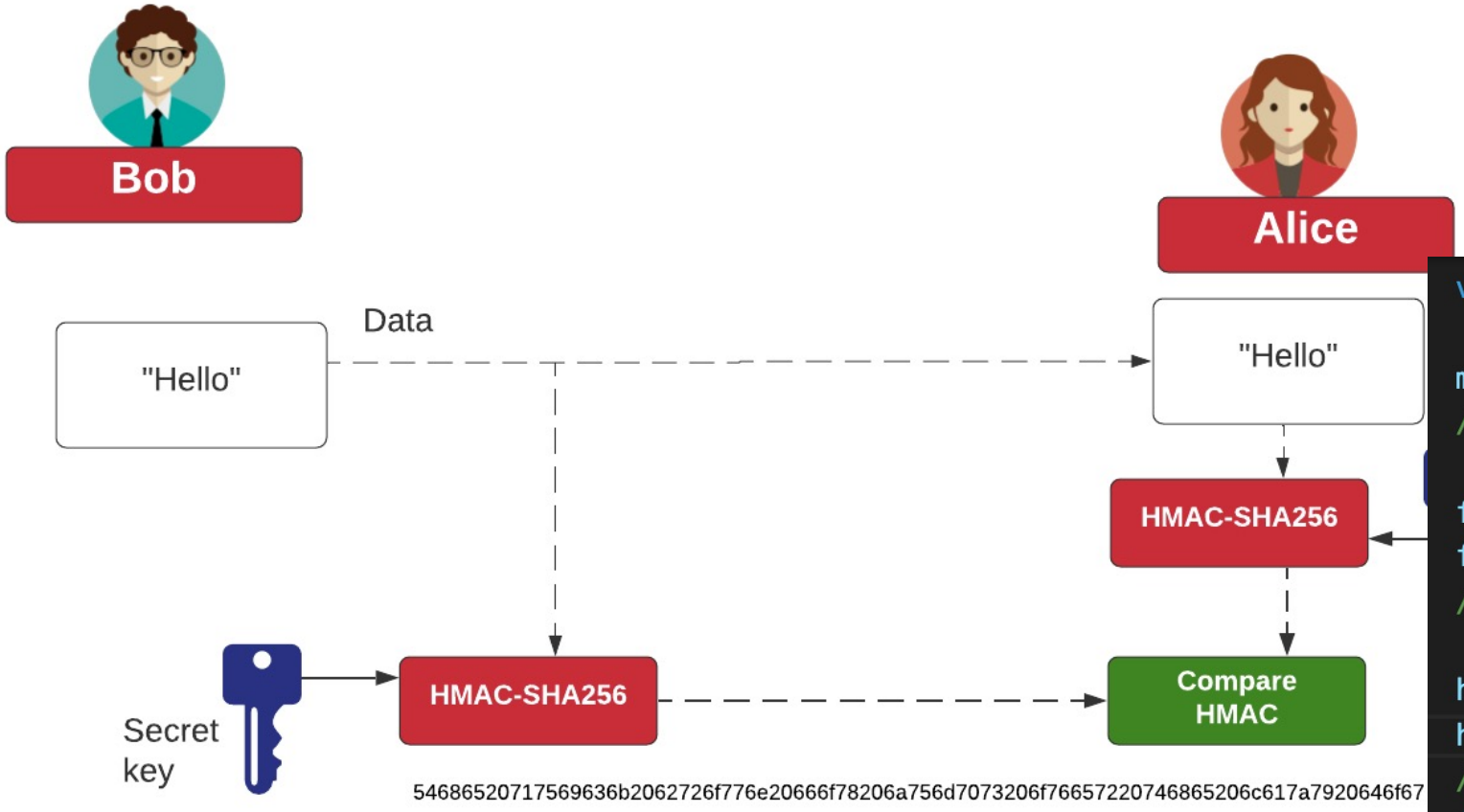


HMAC (Key Hash Message Authentication Code)



https://asecuritysite.com/golang/go_hmac

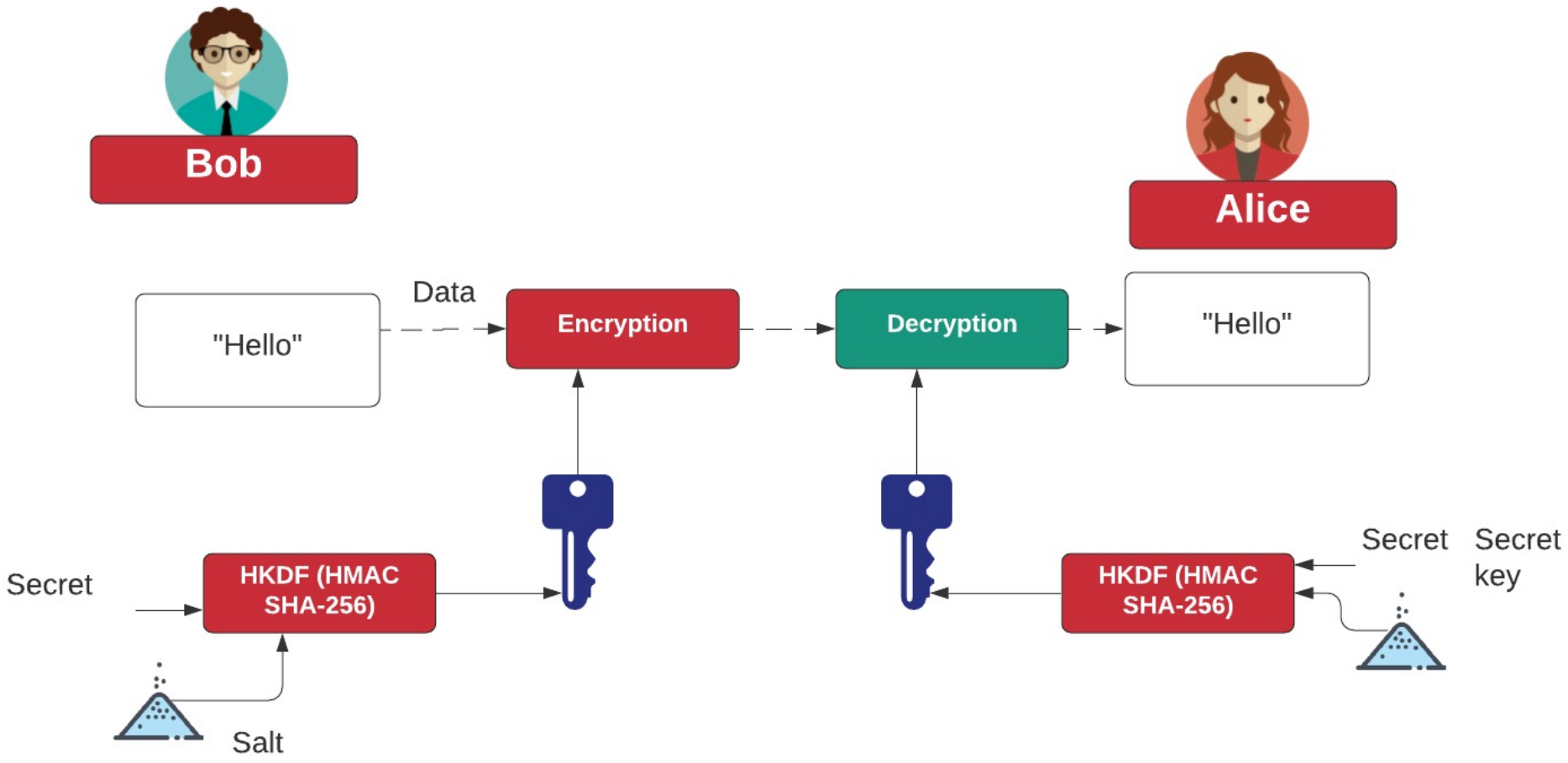
HMAC (Key Hash Message Authentication Code)




```go
var secretKey, _ = hex.DecodeString(key)

message := []byte(m)
//  salt := getSalt(16)

fmt.Printf("Message: %s\n", m)
fmt.Printf("Key: %x\n", secretKey)
//  fmt.Printf("\nSalt: %x\n", salt)

hash := hmac.New(sha256.New, []byte(secretKey))
hash.Write(message)
//  hash.Write(salt)
fmt.Printf("\nHMAC-Sha256: %x", hash.Sum(nil))

hash = hmac.New(sha512.New, []byte(secretKey))
hash.Write(message)
//  hash.Write(salt)
fmt.Printf("\n\nHMAC-sha512: %x", hash.Sum(nil))
```

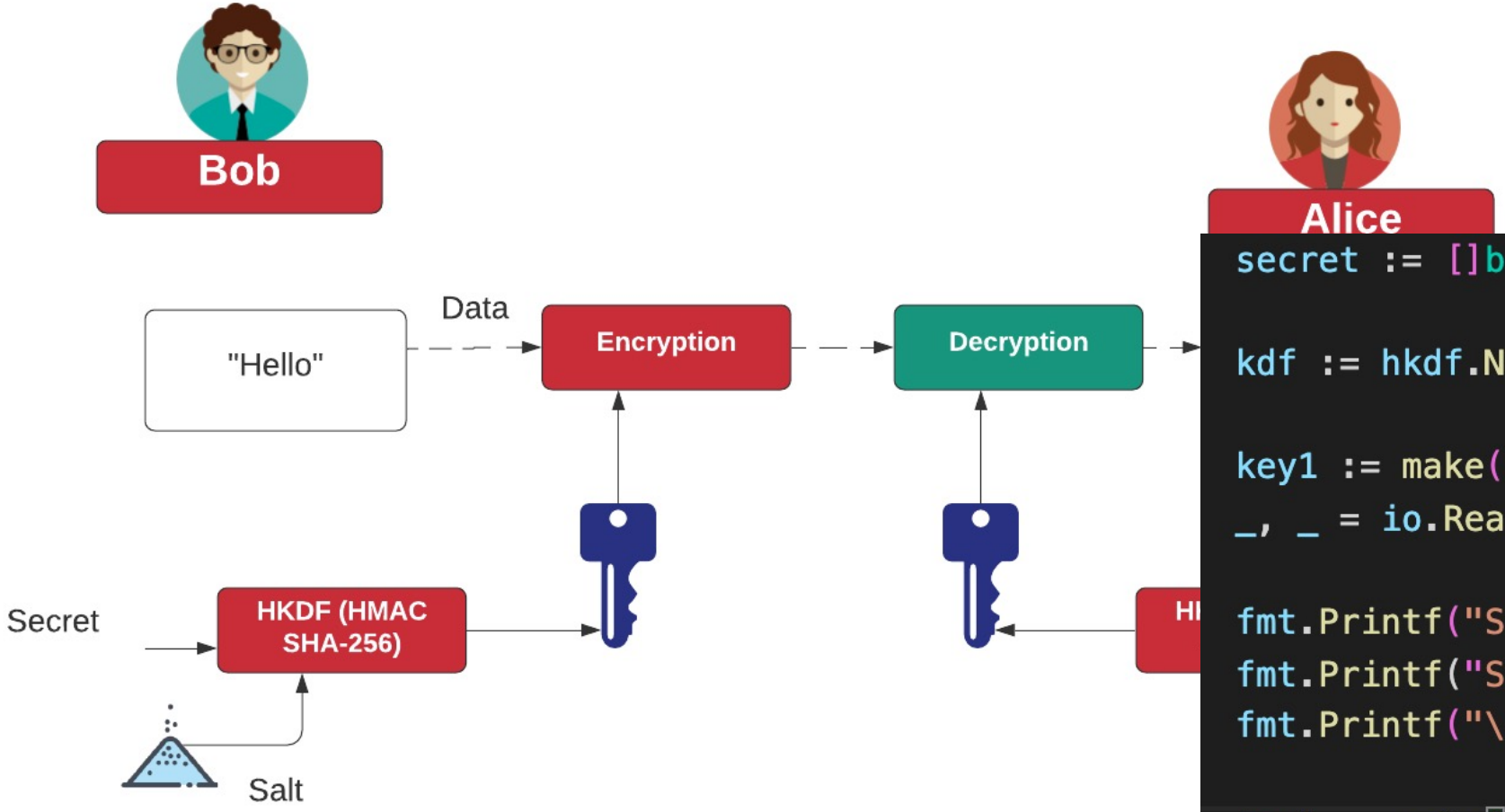
https://asecuritysite.com/golang/go_hmac

HKDF (HMAC Key Derivation Function)



https://asecuritysite.com/golang/go_hkdf

HKDF (HMAC Key Derivation Function)

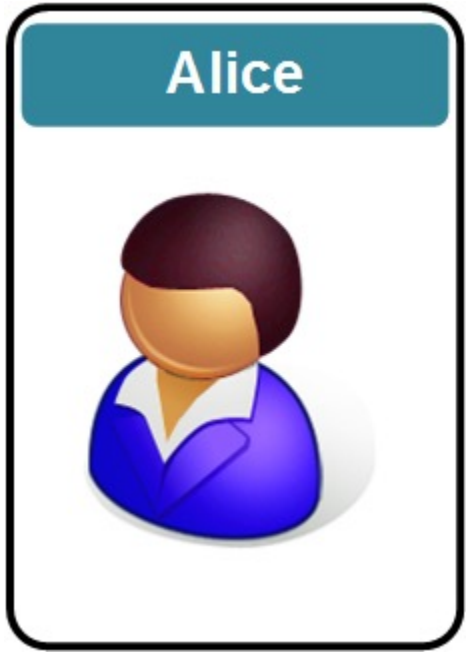



```go
secret := []byte(s)

kdf := hkdf.New(hash, secret, salt, info)

key1 := make([]byte, 16)
_, _ = io.ReadFull(kdf, key1)

fmt.Printf("Secret: %s\n", s)
fmt.Printf("Salt: %x\n", salt)
fmt.Printf("\nHKDF 16 byte key: %x\n", key1)

key2 := make([]byte, 32)
_, _ = io.ReadFull(kdf, key2)

fmt.Printf("HKDF 32 byte key: %x\n", key2)
```

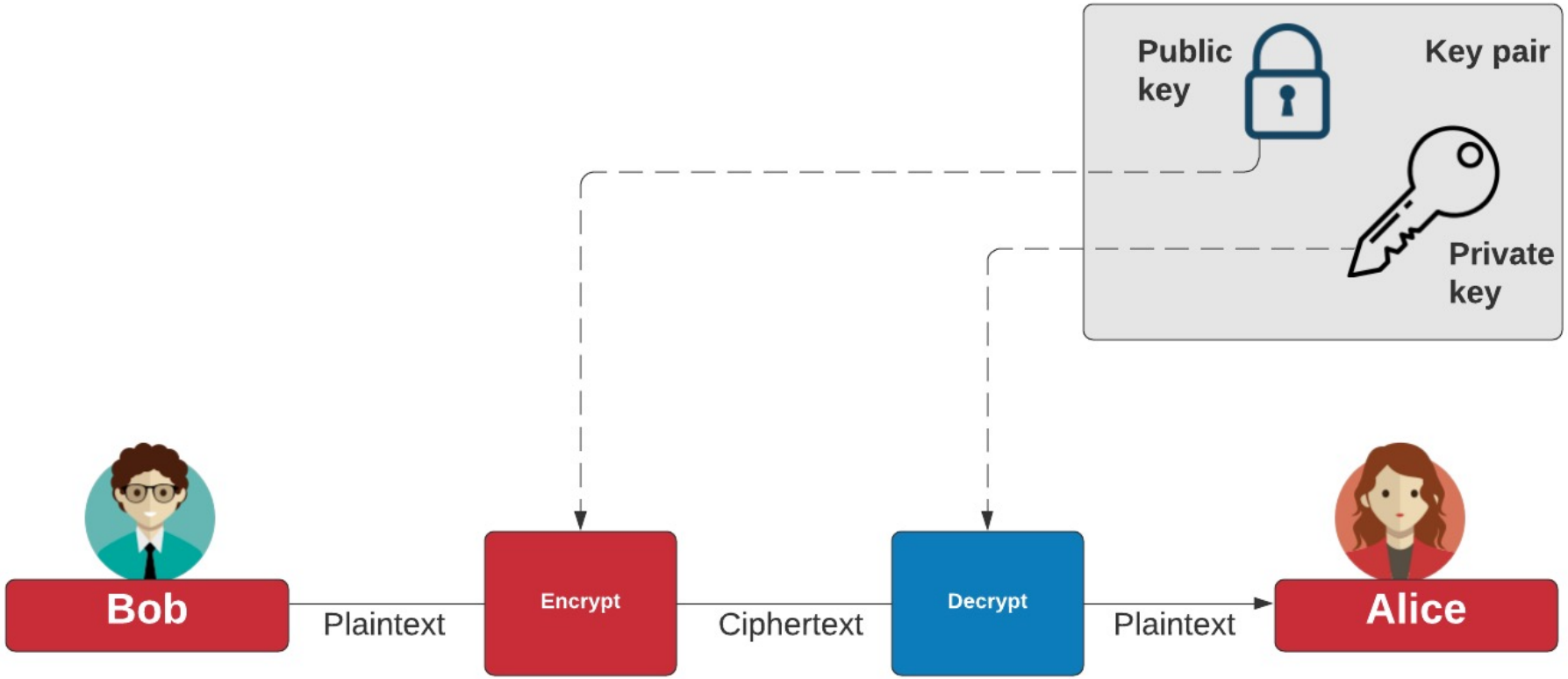
https://asecuritysite.com/golang/go_hkdf

Golang and Crypto: Public Key

Prof Bill Buchanan OBE
https://asecuritysite.com/golang



Golang and Crypto – Public Key



Golang and Crypto – Public Key

Public Key

RSA

Pick p and q (two large primes)

$N = p.q$

$PHI = (p-1)(q-1)$

Pick e (no factors with PHI)

$d = InvMod(e,PHI)$

Public: (e,N)
Private: (d,N)

$C = M^e \pmod{N}$
$P = C^d \pmod{N}$

ECC

Pick random value sk and curve (a, b, G, p, o). G is the base point on curve, p is a prime number, and o is the order of the curve).

For example: $y^2=x^3 + ax + b \pmod{p}$

$Pk = sk.G$

Public key: Pk
Private key: sk

ElGamal

Pick random value x, generator (g) and prime (p)

$Y = g^x \pmod{p}$

Public key: (Y,p)
Private: (x,p)

https://asecuritysite.com/golang/go_elgamal

Golang and Crypto – RSA

RSA

Pick p and q (two large primes)

$N = p.q$

$PHI = (p-1)(q-1)$

Pick e (no factors with PHI)

$d = InvMod(e,PHI)$

Public: (e,N)
Private: (d,N)

$C = M^e \pmod N$
$P = C^d \pmod N$

ECC

Pick random value *sk* and curve (*a*,

Pick ran
(*g*) and

$Y = g^x$ (

Public k
Private:

```go
p, _ := rand.Prime(rand.Reader, psize)
q, _ := rand.Prime(rand.Reader, psize)

M, _ := new(big.Int).SetString(Mval, 10) // Base 10

N := new(big.Int).Mul(p, q)

C := new(big.Int).Exp(M, e, N)

Pminus1 := new(big.Int).Sub(p, new(big.Int).SetInt64(1))
Qminus1 := new(big.Int).Sub(q, new(big.Int).SetInt64(1))
PHI := new(big.Int).Mul(Pminus1, Qminus1)

d := new(big.Int).ModInverse(e, PHI)

Plain := new(big.Int).Exp(C, d, N)
```

https://asecuritysite.com/golang/go_elgamal

Golang and Crypto – ElGamal

Initially Bob creates his public key by selecting a g value and a prime number (p) and then selecting a private key (x). He then computes Y which is:

$$Y = g^x \pmod{p}$$

His public key is (Y, g, p) and he will send this to Alice. Alice then creates a message (M) and selects a random value k). She then computes a and b:

$$a = g^k \pmod{p}$$

$$b = y^k M \pmod{p}$$

Bob then receives these and decrypts with:

$$M = \frac{b}{a^x} \pmod{p}$$

This works because

$$\frac{b}{a^x} \pmod{p} = \frac{y^k M}{(g^k)^x} \pmod{p} = \frac{(g^x)^k M}{(g^k)^x} \pmod{p} == \frac{g^{xk} M}{g^{xk}} \pmod{p} = M$$

https://asecuritysite.com/golang/go_elgamal

Golang and Crypto – ElGamal

Initially Bob creates his public key by selecting a g value and a prime number (p) and then selecting a private key (x). He then computes Y which is:

$$Y = g^x \pmod{p}$$

His public key is (Y, g, p) and he will send this to Alice. Alice then creates a message (M) and selects a random value k). She then computes a and b:

$$a = g^k \pmod{p}$$

$$b = y^k M \pmod{p}$$

Bob then receives these and decry

$$M = \frac{b}{a^x} \pmod{p}$$

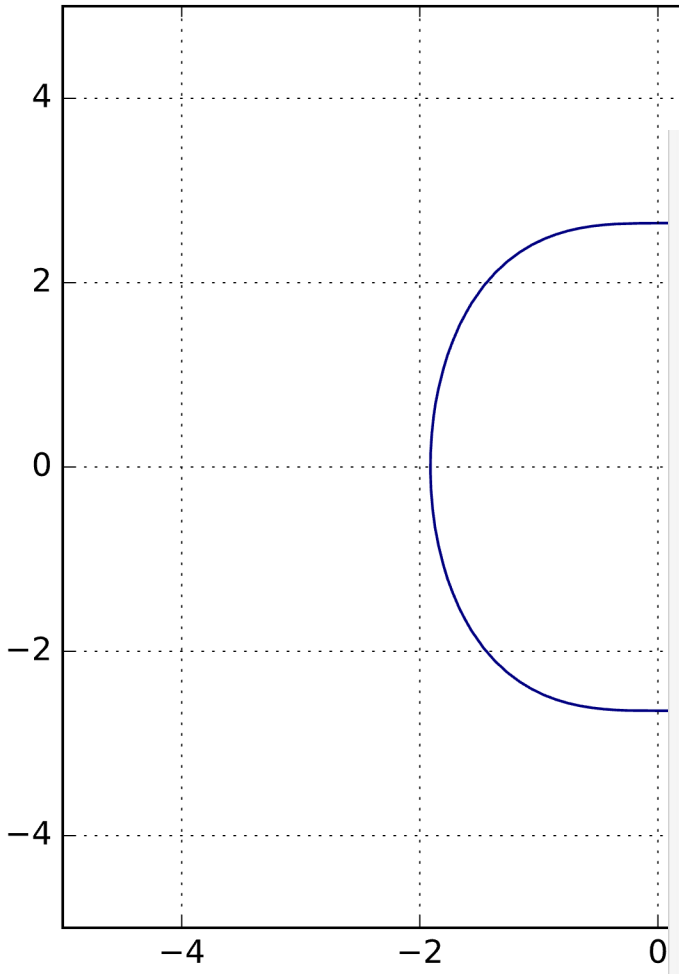
This works because

$$\frac{b}{a^x} \pmod{p} = \frac{y^k M}{(g^k)^x} \pmod{p} = \frac{(g}{}$$

```go
func Encrypt(random io.Reader, pub *PublicKey, msg []byte) (a, b *big.Int, err error) {


    k, _ := rand.Int(random, pub.P)


    m :=new(big.Int).SetBytes(msg)

    a = new(big.Int).Exp(pub.G, k, pub.P)
    s := new(big.Int).Exp(pub.Y, k, pub.P)
    b = s.Mul(s, m)
    b.Mod(b, pub.P)

    return
}
```

https://asecuritysite.com/golang/go_elgama

Golang and Crypto – Elliptic Curve




```
EllipticCurve = collections.namedtuple('EllipticCurve', 'name p a b g n h')

curve = EllipticCurve(
    'secp256k1',
    # Field characteristic.
    p=0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f,
    # Curve coefficients.
    a=0,
    b=7,
    # Base point.
    g=(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,
       0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8),
    # Subgroup order.
    n=0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141,
    # Subgroup cofactor.
    h=1,
)
```

Golang and Crypto – Elliptic Curve

A: 0

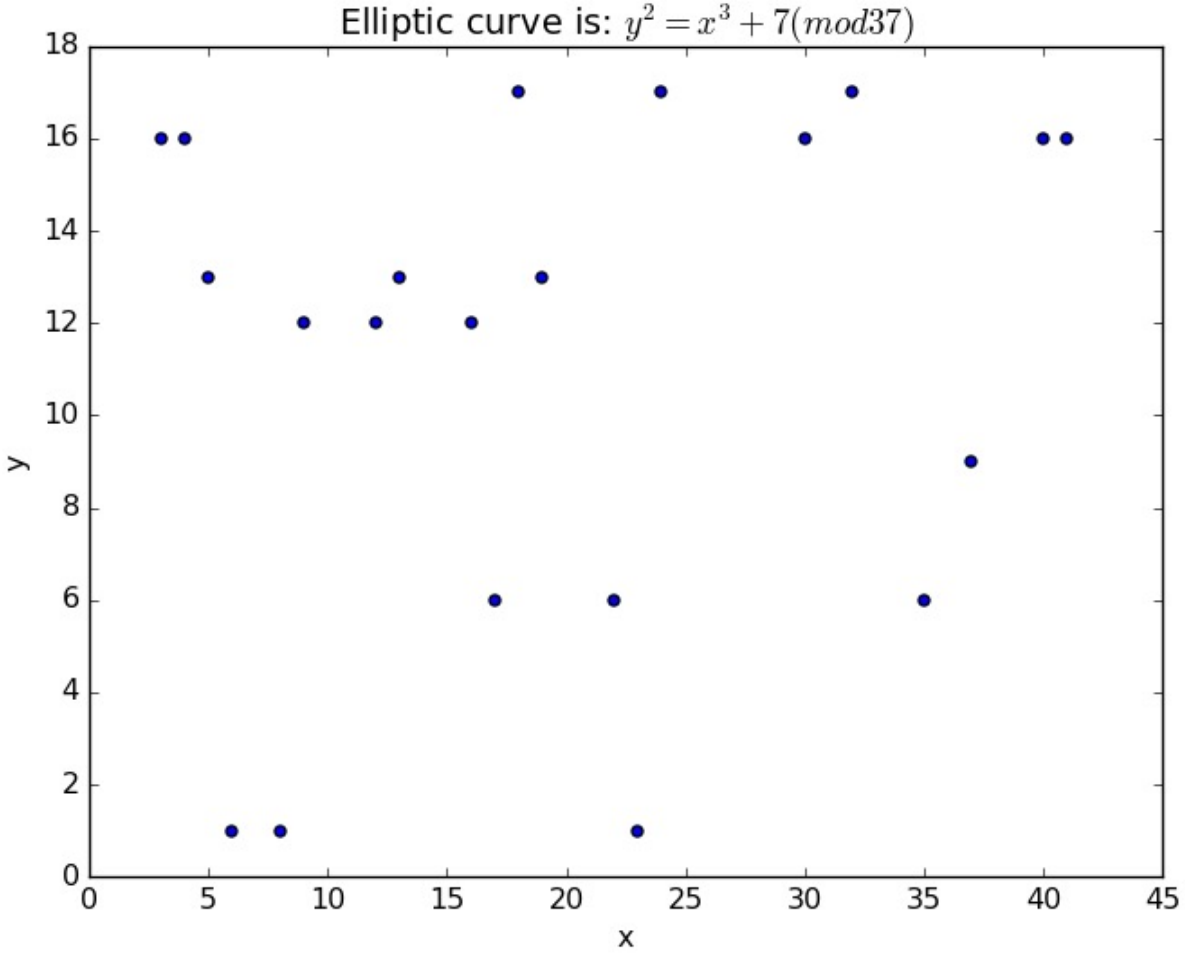
B: 7

Prime number: 37

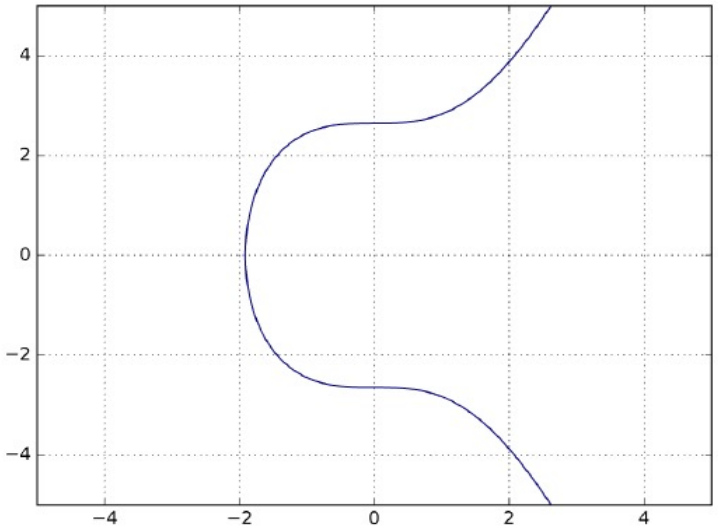
Elliptic curve is: $y^2=x^3+7$

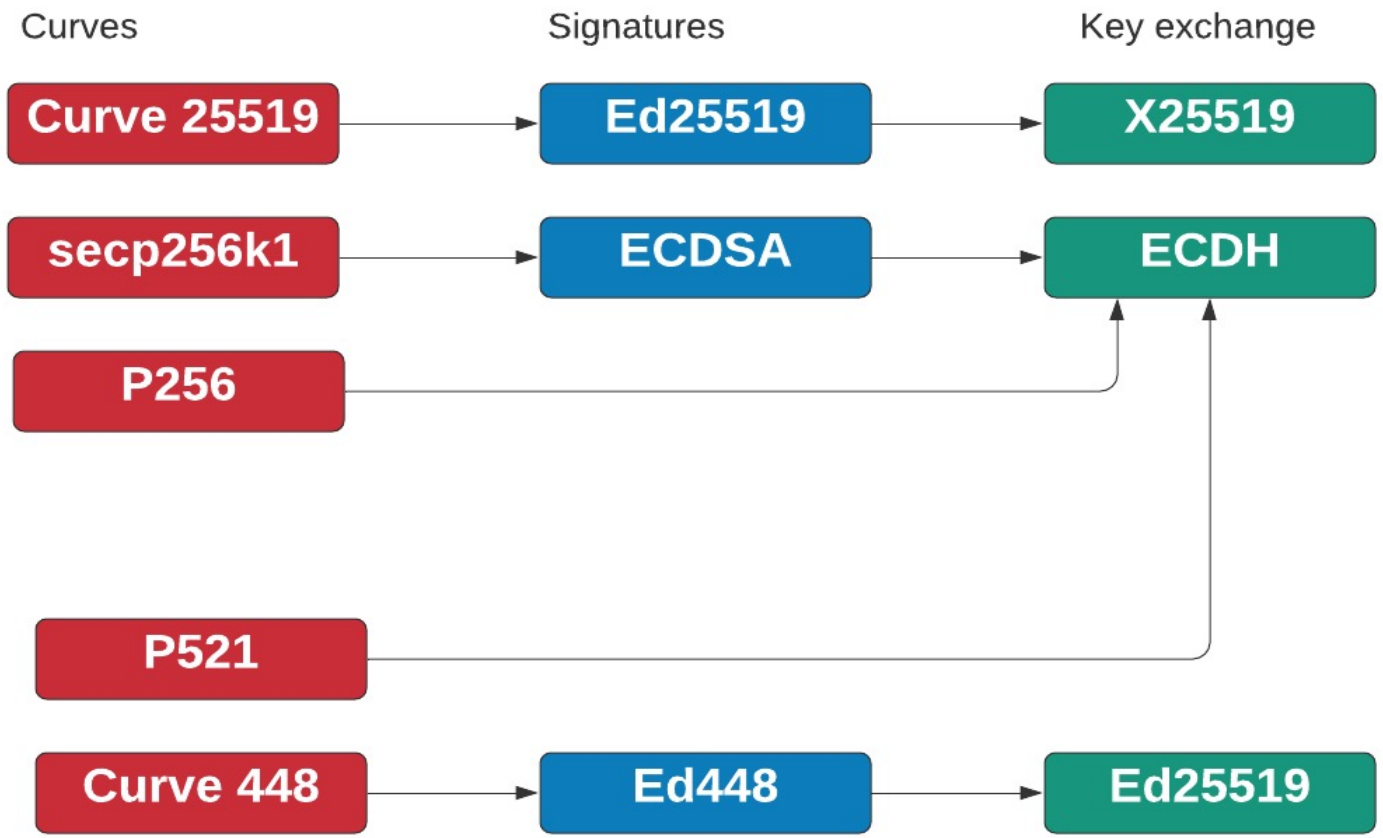
Finding the first 20 points

(3 , 16) (4 , 16) (5 , 13) (6 , 1) (8 , 1) (
, 12) (12 , 12) (13 , 13) (16 , 12) (17 , 6
(18 , 17) (19 , 13) (22 , 6) (23 , 1) (24 ,
17) (30 , 16) (32 , 17) (35 , 6) (37 , 9)
40 , 16) (41 , 16) [Link]



Golang and Crypto – Elliptic Curve Types

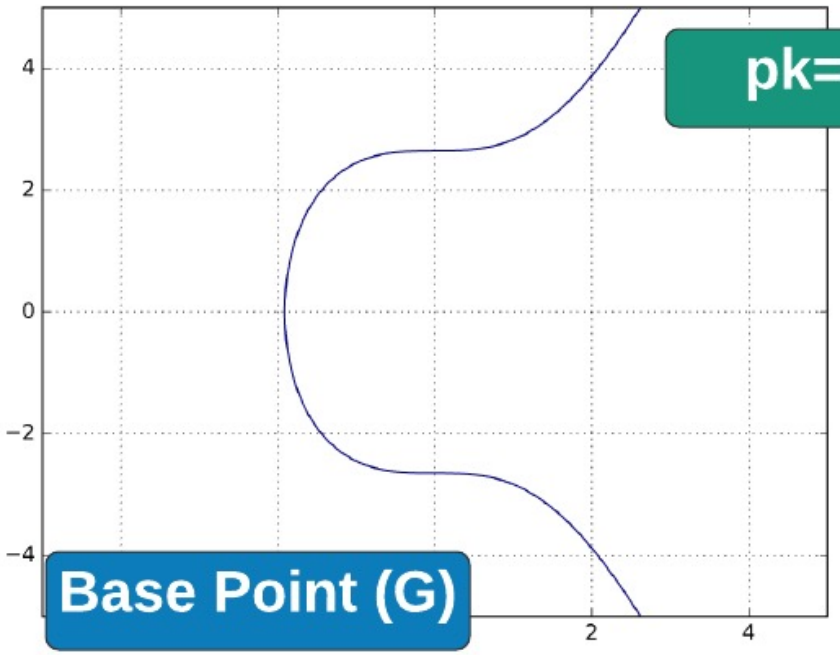




https://asecuritysite.com/ecc/ecc_types

Golang and Crypto – Elliptic Curve Types

$y^2=x^3+ax+b \pmod p$



pk=sk.G

Bob

sk (private key)

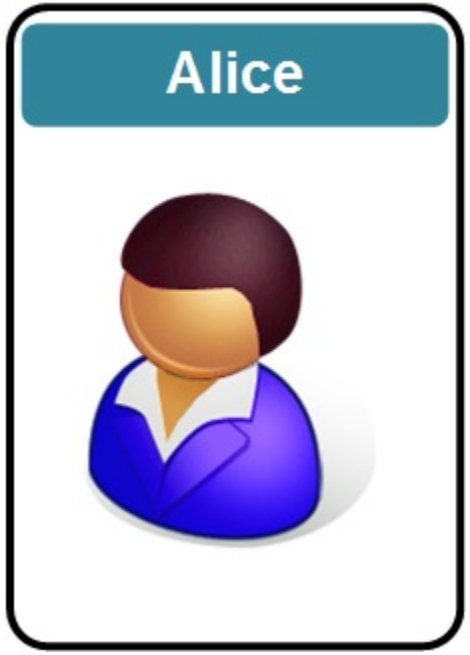
Basic operations:
- Point add.
- Point double.

https://asecuritysite.com/ecc/ecc_types

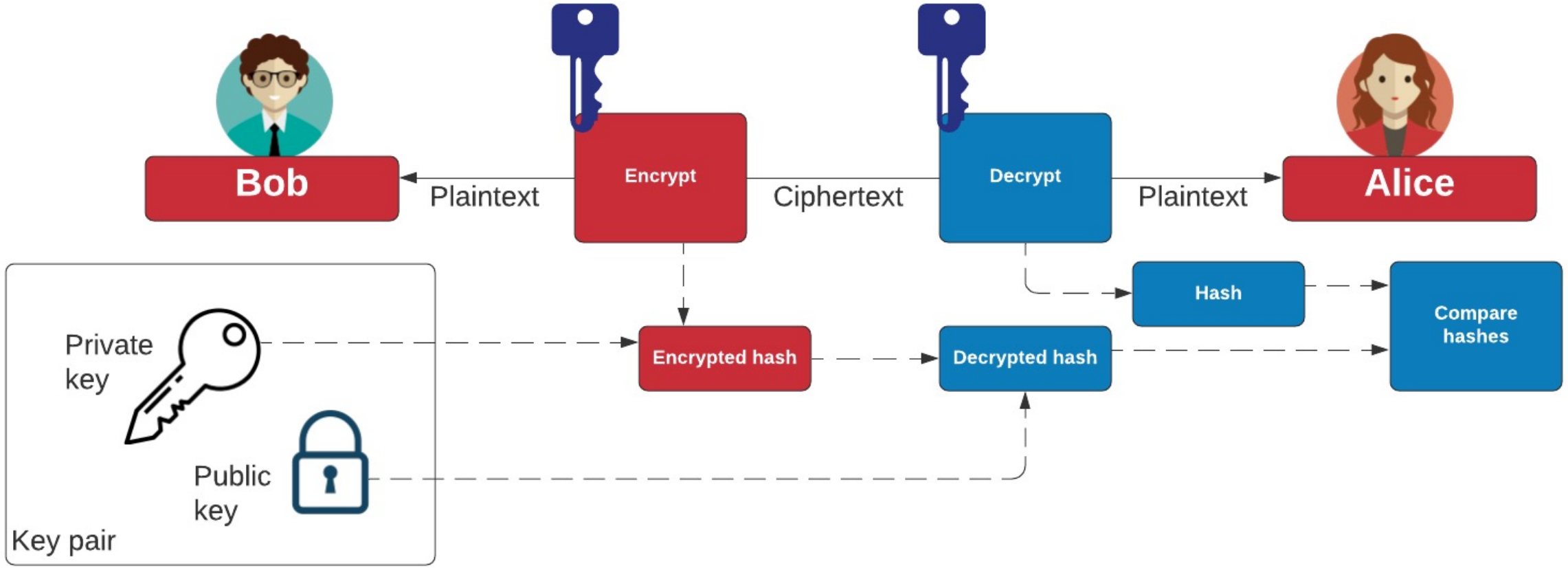
Golang and Crypto: Digital Signatures

Prof Bill Buchanan OBE

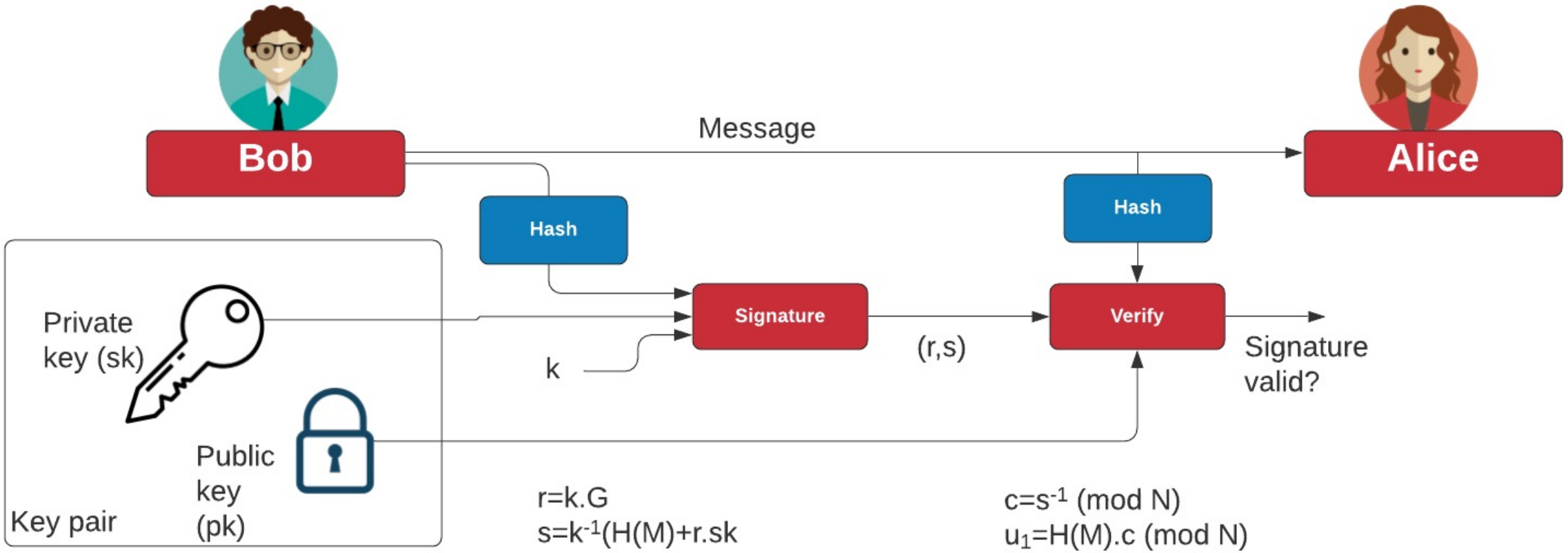
https://asecuritysite.com/golang



Golang and Crypto – Digital Signatures



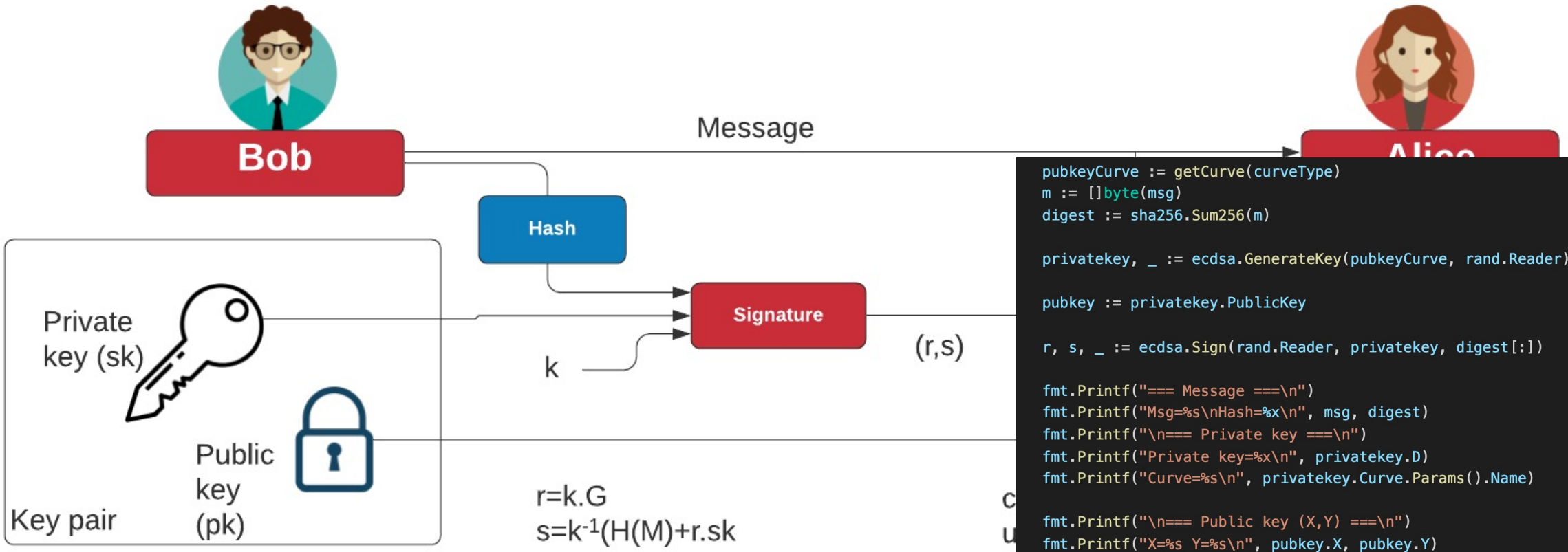
Golang and Crypto – ECDSA



$r=k.G$
$s=k^{-1}(H(M)+r.sk$

$c=s^{-1} \pmod N$
$u_1=H(M).c \pmod N$
$u_2=r.c \pmod N$
$P=u_1.G+u_2.pk$
Prove $r == P \pmod N$

Golang and Crypto – ECDSA




```go
pubkeyCurve := getCurve(curveType)
m := []byte(msg)
digest := sha256.Sum256(m)

privatekey, _ := ecdsa.GenerateKey(pubkeyCurve, rand.Reader)

pubkey := privatekey.PublicKey

r, s, _ := ecdsa.Sign(rand.Reader, privatekey, digest[:])

fmt.Printf("=== Message ===\n")
fmt.Printf("Msg=%s\nHash=%x\n", msg, digest)
fmt.Printf("\n=== Private key ===\n")
fmt.Printf("Private key=%x\n", privatekey.D)
fmt.Printf("Curve=%s\n", privatekey.Curve.Params().Name)

fmt.Printf("\n=== Public key (X,Y) ===\n")
fmt.Printf("X=%s Y=%s\n", pubkey.X, pubkey.Y)
fmt.Printf("  Hex: X=%x Y=%x\n", pubkey.X.Bytes(), pubkey.Y.Bytes())
fmt.Printf("\n=== Signature (R,S) ===\n")
fmt.Printf("R=%s S=%s\n", r, s)
fmt.Printf(" Hex: R=%x S=%x\n", r, s)

rtn := ecdsa.Verify(&pubkey, digest[:], r, s)

if rtn {
    fmt.Printf("Signature verifies")
} else {
    fmt.Printf("Signature does not verify")
}
```

https://asecuritysite.com/golang/ecdsa_rs2

https://asecuritysite.com/golang/ecdsa_rs

Golang and Crypto – EdDSA

With Ed25519, Bob first generates a 256-bit random number for his private key ($priv$) and then create the public key with:

$$pub = H(priv)[:32] \cdot B$$

and where B is the base point on the curve, and where $H()$ is a hash function (SHA-512). In this case we take the lower 32 bytes of the SHA-512 hash. Normally we just define this as the y co-ordinate of the point. The public key is thus 32 bytes long (256 bits). To compute the signature, we generate:

$$r = H_{int}(H(priv)[32:] \,||\, M) \pmod{l}$$

and where $H_{int}()$ is a hash function (SHA-512) and converted to an integer value. In this case we take the upper 32 bytes for the calculation. The value of l is the order of the curve. Then M is the byte value of the message. With "||" we append the bytes together. We then convert this to a point on the curve with:

$$R = r \cdot B$$

Next, we compute:

$$h = H_{int}(R \,||\, pub \,||\, M) \pmod{l}$$

and:

$$s = H_{int}(priv))[:32]$$

$$S = (r + h \cdot s) \pmod{l}$$

The signature is then (R, S). Bob sends M, R, S and pub to Alice. Alice then computes:

$$k = H(R \,||\, pub \,||\, M) \pmod{l}$$

$$P_1 = S \cdot B$$

$$P_2 = R + k \cdot pub$$

and if P_1 is equal to P_2, the signature verifies. This works because:

$$P_1 = S \cdot B = (r + h \cdot s \pmod{l}) \cdot B = r \cdot B + h \cdot s \cdot B = R + h \cdot pub = P_2$$

https://asecuritysite.com/golang/ed25519

Golang and Crypto – EdDSA

With Ed25519, Bob first generates a 256-bit random number for his private key ($priv$) and then create the public key with:

$$pub = H(priv)[:32] \cdot B$$

and where B is the base point on the curve, and where $H()$ is a hash function (SHA-512). In this case we take the lower 32 bytes of the SHA-512 hash. Normally we just define this as the y co-ordinate of the point. The public key is thus 32 bytes long (256 bits). To compute the signature, we generate:

$$r = H_{int}(H(priv)[32:] \,||\, M) \pmod{l}$$

and where $H_{int}()$ is a hash function (SHA-512) and converted to an integer value. In this case we take the upper 32 bytes for the calculation. The value of l is the order of the curve. Then M is the byte value of the message. With "||" we append the bytes together. We then convert this to a point on the curve with:

$$R = r \cdot B$$

Next, we compute:

$$h = H_{int}(R \,||\, pub \,||\, M) \pmod{l}$$

and:

$$s = H_{int}(priv))[:32]$$

$$S = (r + h \cdot s) \pmod{l}$$

The signature is then (R, S). Bob sends M, R, S and pub to Alice. Alice then con

$$k = H(R \,||\, pub \,||\, M) \pmod{l}$$

$$P_1 = S \cdot B$$

$$P_2 = R + k \cdot pub$$

and if P_1 is equal to P_2, the signature verifies. This works because:

$$P_1 = S \cdot B = (r + h \cdot s \pmod{l}) \cdot B = r \cdot B + h \cdot s \cdot B = R + h \cdot pub = P_2$$

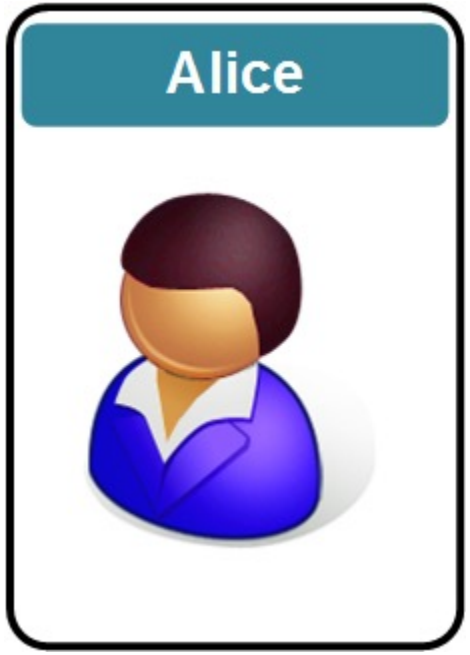
```go
publ, priv, _ := ed25519.GenerateKey((nil))

m := []byte(msg)
digest := sha256.Sum256(m)

sig := ed25519.Sign(priv, digest[:])

fmt.Printf("=== Message ===\n")
fmt.Printf("Msg=%s\nHash=%x\n", msg, digest)
fmt.Printf("\n=== Private key ===\n")
fmt.Printf("Public key=%x\n\n", publ)
fmt.Printf("Private key=%x\n\n", priv[0:32])
fmt.Printf("Signature: (%x,%x)\n\n", sig[0:32], sig[32:64])

rtn := ed25519.Verify(publ, digest[:], sig)
```

https://asecuritysite.com/golang/ed25519

Golang and Crypto 1:

Prof Bill Buchanan OBE

https://asecuritysite.com/golang

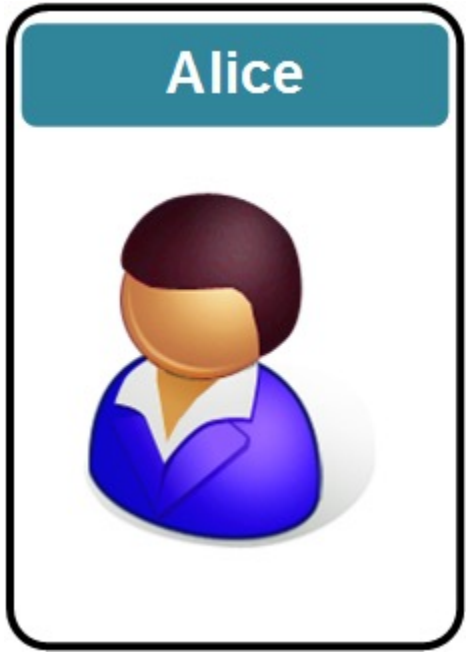


Coverage:

1. Formats: Hex, Base64 and Big Integers.
2. Symmetric Key: Block Stream and AEAD.
3. Hashing.
4. HMAC and HKDF.
5. Public Key: RSA, ElGamal and ECC.
6. Digital Signatures: ECDSA and EdDSA.

Golang and Crypto 2:

Prof Bill Buchanan OBE
https://asecuritysite.com/golang



Coverage:

1. Shamir Secret Shares (SSS).
2. Verifiable Secret Shares (VSS).
3. Threshold signatures.
4. Crypto Pairs.
5. Zero Knowledge Proofs.
6. Homomorphic Encryption.